УДК 004.42
ББК 32.973
О 75

Автор-составитель О. И. Еськова, канд. техн. наук, доцент

Рецензенты: В. Д. Левчук, канд. техн. наук, доцент, зав. кафедрой автоматизированных систем обработки информации Гомельского государственного университета им. Ф. Скорины;
В. В. Бондарева, канд. техн. наук, доцент кафедры информационно-вычислительных систем Белорусского торгово-экономического университета потребительской кооперации

Рекомендовано к изданию научно-методическим советом учреждения образования «Белорусский торгово-экономический университет потребительской кооперации». Протокол № 2 от 13 декабря 2016 г.

О 75 **Основы** алгоритмизации и программирования : пособие для реализации содержания образовательных программ высшего образования I ступени. В 2 ч. Ч. 1 / авт.-сост. О. И. Еськова. – Гомель : учреждение образования «Белорусский торгово-экономический университет потребительской кооперации», 2018. – 116 с.
ISBN 978-985-540-429-4

Пособие предназначено для студентов специальности 1-28 01 01 «Экономика электронного бизнеса». Содержит теоретический материал и практические задания, решение которых поможет выработать умения и навыки самостоятельной разработки программ.

УДК 004.42
ББК 32.973

ISBN 978-985-540-429-4 (ч. 1)
ISBN 978-985-540-430-0

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Дисциплина «Основы алгоритмизации и программирования» является первой в цикле специальных дисциплин обучения технологиям программирования в рамках специальности 1-28 01 01 «Экономика электронного бизнеса». Ее основная цель – развить алгоритмическое мышление студентов, познакомить их с типами и структурами данных, различными парадигмами программирования и сформировать практические навыки программирования процедурно-ориентированных алгоритмов.

В качестве первого языка обучения программированию в последние годы обычно используют язык программирования С («Си»). Этот язык прошел значительную проверку временем и доказал свою жизнеспособность. Он содержит все ключевые возможности процедурно-ориентированных языков (управляющие конструкции, типы и структуры данных), что позволяет использовать его не только для целей промышленной разработки, но и целей обучения. При этом язык С близок к машинно-ориентированным языкам программирования, допуская программиста ко всем «низкоуровневым» ресурсам компьютера, чего не позволяют такие языки, как, например, Паскаль и Бейсик. Это, с точки зрения автора, не является препятствием для его использования в качестве первого языка обучения, поскольку позволяет студентам лучше понять устройство компьютера, основы представления данных и процесса выполнения компьютерных программ. Немаловажным фактором для выбора именно этого языка программирования является и то, что язык С оказал большое влияние на многие другие современные языки программирования (C++, C#, Java), которые объединяет схожий синтаксис и общее название «С-подобные языки».

Данное издание ориентировано на среду разработки Microsoft Visual Studio и затрагивает также некоторые аспекты языка программирования C++. Эти аспекты не касаются парадигмы объектно-ориентированного программирования и являются лишь некоторым «усовершенствованием» языка С, позволяющим дополнить и раскрыть его возможности.

Предлагаемое пособие состоит из двух частей, каждая из которых соответсвует семестру изучения дисциплины. Настоящее издание включает первую часть, которая охватывает пять тем от введения в язык программирования С и среду Microsoft Visual Studio до работы с массивами. Теория алгоритмизации представлена в пособии в констексте использования языка программирования С, поэтому соответсвующие вопросы распределены по темам пособия. Так, в теме 1 раскрывается понятие алгоритма, его свойства, способы представления и основные виды. В теме 4 рассматриваются парадигма структурного программирования и принцип нисходящего проектирования программ. В теме 5 описываются основные алгоритмы сортировки и поиска. В целом материал пособия представлен таким образом, чтобы понятия и возможности языка программирования раскрывались постепенно, по принципу «от простого – к сложному».

Каждая тема пособия включает задачи, решение которых позволит сформировать у студентов навыки самостоятельной разработки программ.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ, ЗАДАЧИ ДЛЯ ПРАКТИЧЕСКИХ ЗАНЯТИЙ

ТЕМА 1. ВВЕДЕНИЕ В ЯЗЫК ПРОГРАММИРОВАНИЯ С

1.1. Понятие алгоритма. Способы описания алгоритмов

Алгоритмом называется точное и понятное предписание исполнителю совершить последовательность действий, направленных на решение поставленной задачи.

Слово «алгоритм» происходит от имени математика Аль-Хорезми, который сформулировал правила выполнения арифметических действий. Первоначально под алгоритмом понимали только эти правила выполнения четырех арифметических действий над числами. В дальнейшем это понятие стали использовать для обозначения любой последовательности действий, приводящих к решению поставленной задачи.

Основными *свойствами* алгоритма являются:

• *Понятность и выполнимость* для исполнителя. Например, предписание «Взвейтесь кострами, синие ночи!» не является алгоритмом, поскольку «его исполнитель» (синие ночи) не может выполнить соответствующее действие. Для алгоритмов, которые будут здесь изучаться, исполнителем является компьютер. Поэтому алгоритм должен быть записан на языке, понятном компьютеру (языке программирования). В то же время алгоритм должен быть понятен и человеку, который его анализирует. Для этого существуют формы записи алгоритма, ориентированные на человека (словесное описание, блок-схема и псевдокод).

• *Детерминированность (определенность).* Каждое правило алгоритма должно быть четким и однозначным. Это свойство обеспечивает выполнение алгоритма механически, не требуя никаких дополнительных указаний или сведений о решаемой задаче.

• *Результативность.* Алгоритм должен приводить к решению задачи за конечное число шагов.

• *Массовость*. Это свойство предполагает, что алгоритм должен быть пригоден для решения некоторого класса задач, которые различаются только исходными данными.

• *Дискретность.*Алгоритм должен представлять процесс решения задачи как последовательное исполнение простых (или ранее определенных) шагов (этапов).

Основными *способами описания* алгоритма являются текст программы на языке программирования, словесное описание, блок-схема и псевдокод.

Словесный способ записи алгоритмов является описанием последовательных этапов обработки данных. Алгоритм может быть задан в произвольном изложении на естественном языке. Например, алгоритм нахождения наибольшего общего делителя двух натуральных чисел можно представить как следующую последовательность действий (шагов):

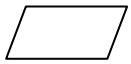
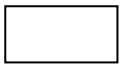
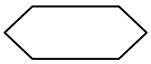
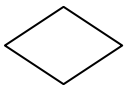
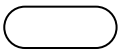
1. Задание двух чисел.
2. Если числа равны, то выбор любого из них в качестве ответа и остановка, в противном случае – продолжение выполнения алгоритма.
3. Определение большего из чисел.
4. Замена большего из чисел разностью большего и меньшего.
5. Повтор алгоритма с шага 2.

Приведенный алгоритм используется для любых натуральных чисел и должен приводить к решению поставленной задачи.

В *блок-схеме* каждый из типов действий (ввод исходных данных, вычисление значений выражений, проверка условий, управление повторением действий, окончание обработки и т. п.) соответствует геометрической фигуре, представленной в виде блочного символа. Блочные символы соединены линиями переходов, которые определяют очередность выполнения действий. Примеры наиболее распространенных блочных символов и их интерпретация приведены в таблице 1.1.

Псевдокод является системой обозначений и правил, которая предназначена для единообразной записи алгоритмов. Он занимает промежуточное место между естественным и формальным языками. С одной стороны, псевдокод похож на обычный естественный язык, поэтому алгоритмы могут на нем записываться и читаться как обычный текст. С другой стороны, в псевдокоде используются некоторые формальные конструкции и математическая символика, благодаря чему запись алгоритма приближается к общепринятой математической записи.

Таблица 1.1 – **Основные блочные символы**

Символ	Название	Назначение
	Данные	Общее обозначение ввода или вывода данных
	Процесс	Обработка данных, операция или группа операций
	Соединитель	Соединение прерванных линий потока
	Предопределенный процесс	Вычисления по подпрограмме (модулю)
	Подготовка	Задание изменений параметров цикла
	Решение	Проверка условия
	Терминатор	Вход или выход во внешнюю среду
	Комментарий	Для записи пояснений к алгоритму

В псевдокоде, так же как и в языках программирования, существуют служебные слова, смысл которых определен раз и навсегда. Их выделяют в печатном тексте жирным шрифтом, а в рукописном тексте подчеркивают. (Примеры использования псевдокода см. на рисунках 1.2 и 1.6.) Единого формального подхода к определению псевдокода не существует, поэтому используются различные его варианты, отличающиеся набором служебных слов и основных (базовых) конструкций.

Можно выделить три основных вида алгоритмов:

- линейный;
- ветвящийся;
- циклический.

Все остальные алгоритмы получаются путем сочетания (композиции) этих базовых видов.

Линейным называется такой алгоритм, при котором все этапы решения задачи выполняются последовательно.

Пример 1.1. Алгоритм расчета длины окружности.

Поскольку алгоритм должен быть массовым, он должен работать для окружностей любого радиуса. Первым шагом алгоритма, следовательно, будет ввод конкретного радиуса окружности. Введенное значение нужно где-то хранить в памяти. Для хранения данных в любом языке программирования используется понятие переменной.

Переменная – это некоторая область памяти, которая имеет свое имя. Запись числа в переменную называется присваиванием значения и обозначается в С знаком «=». Старое значение переменной при этом теряется (затирается).

В этом алгоритме нам нужны две переменные: одна для хранения радиуса (ее имя *r*), а другая – для хранения длины окружности (*L*).

После ввода радиуса в переменную *r* нужно рассчитать длину окружности и записать ее в переменную *L*.

Последний шаг алгоритма – вывод результата на экран компьютера. Блок-схема алгоритма показана на рисунке 1.1.

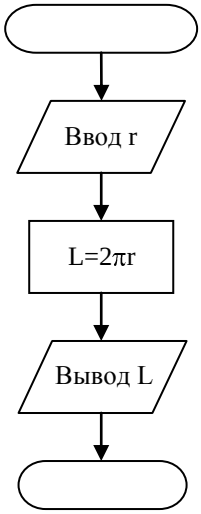


Рисунок 1.1 – **Блок-схема алгоритма расчета длины окружности**

Тот же алгоритм, записанный на псевдокоде, напоминает естественный язык (рисунок 1.2). Обратите внимание на сдвиг основного текста алгоритма относительно операторных скобок (**Начало**, **Конец**).

Начало
Ввести радиус в переменную r;
L = 2πr;
Вывести L на экран;
Конец

Рисунок 1.2 – **Алгоритм расчета длины окружности на псевдокоде**

Ветвящимся называется такой алгоритм, в котором ход выполнения обработки информации зависит от истинности тех или иных условий.

Пример 1.2. Алгоритм определения максимального из двух чисел.

Первым шагом алгоритма будет ввод конкретных значений чисел *a* и *b*. Затем эти числа нужно сравнить. Если *a* окажется больше *b*, то нужно вывести на экран значение *a*. В противном случае вывести нужно значение *b*.

Блок-схема этого алгоритма показана на рисунке 1.3.

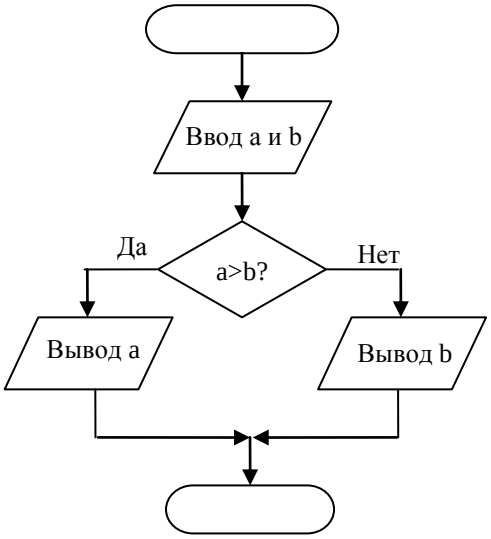


Рисунок 1.3 – **Блок-схема алгоритма определения максимального из двух чисел**

Алгоритм можно модифицировать, если ввести дополнительную переменную *max*, в которую нужно записать значение наибольшего из двух чисел, а затем вывести на экран содержимое ячейки *max* (рисунок 1.4). На первый взгляд этот алгоритм не отличается от первого варианта. Однако его гораздо легче модифицировать на случай трех, четырех и так далее исходных чисел.

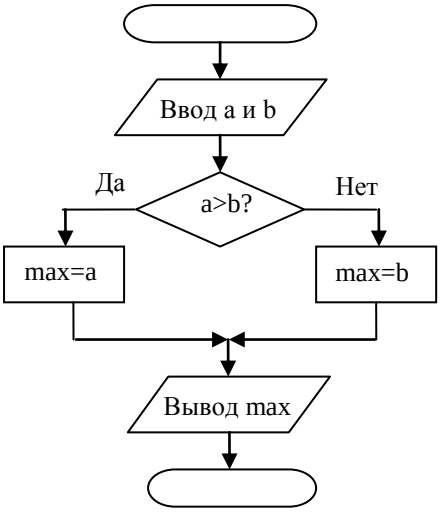


Рисунок 1.4 – **Второй вариант блок-схемы определения максимального из двух чисел**

Допустим теперь, что нужно найти наибольшее из трех чисел: *a, b* и *c*. После ввода конкретных значений этих чисел сравним два первых числа и присвоим результат переменной *max* (как в предыдущем случае).

После этого можно сравнить значение переменной *max* (которая уже содержит наибольшее из двух чисел) с третьим числом *c*. В случае, если значение *c* окажется большим переменной *max*, то можно изменить значение переменной *max*, сделав его равным *c*. В противном случае ничего не нужно делать, и значение *max* не изменится.

После второй проверки переменная *max* будет содержать значение наибольшего из трех чисел, которое и следует вывести на экран. Блок-схема этого алгоритма показана на рисунке 1.5, а аналогичный текст на псевдокоде – на рисунке 1.6.

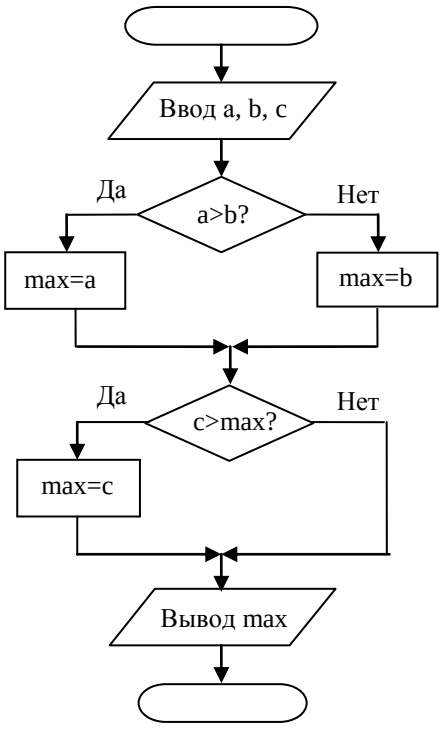


Рисунок 1.5 – **Блок-схема алгоритма определения максимума из трех чисел**

```
Начало
    Ввести a, b и c;
    Если a>b,
        то
            max=a;
        иначе
            max=b;
    Все-если
    Если c>max,
        то
            max=c;
    Все-если
Конец
```

Рисунок 1.6 – **Алгоритм определения максимума из трех чисел на псевдокоде**

Следует заметить, что алгоритм ветвления может иметь сокращенный вид, когда действия по ветке **нет** (или **иначе**) отсутствуют.

Циклическим называется алгоритм, в котором какие-либо действия повторяются многократно. По количеству шагов выполнения циклы делятся на циклы с определенным (заранее заданным) числом повторений и циклы с неопределенным числом повторений.

Если число повторений заранее неизвестно, то в цикле должно проверяться какое-то условие, задающее необходимость выполнения цикла. При этом условие может проверяться в начале цикла – тогда речь идет о цикле с *предусловием*, или в конце – тогда это цикл с *постусловием*.

Пример 1.3. Алгоритм определения суммы целых чисел от *a* до *b*.

Первым шагом этого алгоритма будет ввод конкретных значений *a* и *b*. Вообще, в программах всегда нужно проверять корректность вводимых данных. Но пока для простоты предположим, что пользователь введет числа правильно: *a* будет меньше *b*.

Для того чтобы перебрать все числа от *a* до *b*, введем вспомогательную переменную *x*. Сначала присвоим ей значение *a* и в процессе перебора будем увеличивать на 1. Таким образом, переменная *x* на следующем шаге цикла станет равна *a* + 1, затем *a* + 2 и т. д. Процесс нужно остановить тогда, когда при очередном увеличении «текущего» числа *x* оно станет больше *b*.

Для подсчета суммы введем переменную *S*, которой сначала присвоим 0, т. е. «очистим» ее. На каждом шаге цикла будем прибавлять к *S* очередное значение *x*, чтобы сумма постепенно «накапливалась». Когда перебор чисел от *a* до *b* закончится, переменная *S* будет содержать искомое значение. Останется только вывести это значение на экран компьютера.

Блок-схема этого алгоритма показана на рисунке 1.7, а запись на псевдокоде – на рисунке 1.8.

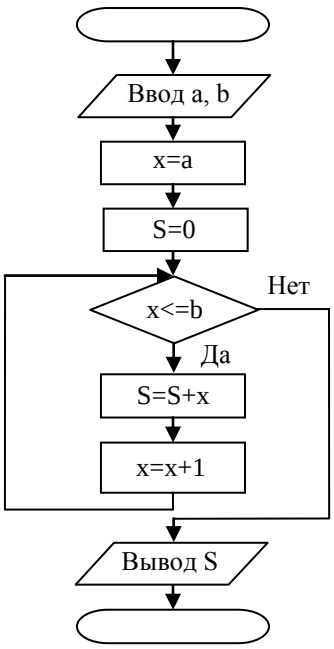


Рисунок 1.7 – **Блок-схема определения суммы чисел от *a* до *b***

```
Начало
    Ввод a, b;
    x=a;
    S=0;
    Пока x<=b
      нц
        S=S+x;
        x=x+1;
      кц
    Вывод S;
Конец
```

Рисунок 1.8 – **Псевдокод алгоритма определения суммы чисел от *a* до *b***

Хотя псевдокод и записан на русском языке, это уже почти программа. Обратите внимание на то, что *тело цикла* (повторяемые действия) записано со сдвигом вправо. Аналогично сдвигаются вправо на 3–4 символа и действия, выполняемые по веткам **Если** или **Иначе**. Такая запись называется *структурной*. Структурной записи обязательно нужно придерживаться при написании программ на любых языках программирования.

1.2. Позиционные системы счисления. Перевод из одной системы счисления в другую

Обычные люди привыкли пользоваться десятичной системой счисления. В компьютере данные представляются в двоичной системе, а программисты зачастую используют шестнадцатеричную систему. Но все эти способы записи чисел относятся к одному виду – *позиционным* системам счисления. В них значение цифры зависит от ее положения (разряда) в числе.

Количество различных цифр, которые можно использовать для записи числа, называется ее *основанием*. Так, если основание системы счисления n, то имеется всего n цифр от 0 до $n – 1$. В десятичной системе используется 10 цифр: 0, 1, 2, …, 9. В двоичной системе – две цифры: 0 и 1. В шестнадцатеричной применяется 16 цифр: 0–9, A–F (т. е. A соответствует десятичной 10, B – 11, …, F–15).

Позиции цифр в числе можно мысленно пронумеровать справа налево, начиная с 0, т. е. крайняя правая позиция имеет номер 0, левее ее – номер 1, еще левее – номер 2 и т. д.

Значение числа получается как сумма произведений цифры на основание системы счисления в степени «номер позиции»:

$$Число = \sum (цифра \cdot основание\ системы^{номер\ позиции}).$$

Пользуясь этим правилом, можно переводить число из любой системы в десятичную.

Например:

- $235_{10} = 2 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0$;
- $11001_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 16 + 8 + 1 = 25_{10}$;
- $267_8 = 2 \cdot 8^2 + 6 \cdot 8^1 + 7 \cdot 8^0 = 128 + 48 + 7 = 183_{10}$;
- $3A2_{16} = 3 \cdot 16^2 + 10 \cdot 16^1 + 2 \cdot 16^0 = 768 + 160 + 2 = 930_{10}$.

Для перевода в обратном направлении (из десятичной системы в любую другую) нужно выполнять целочисленное деление исходного числа на основание той системы счисления, в которую нужно перевести число. При этом важен остаток от деления и частное. Частное делится на основание системы счисления до тех пор, пока не будет получен 0. После этого все остатки нужно выписать в обратном порядке – это и будет число в новой системе счисления.

Пример 1.4. Перевод числа 25 из десятичной системы счисления в двоичную будет выглядеть так, как показано на рисунке 1.9. Результат в этом случае будет следующим: $25_{10} = 11001_2$.

Перевод числа 393 из десятичной в шестнадцатеричную систему требует последовательного деления на число 16 (рисунок 1.10). В результате получим: $393_{10} = 189_{16}$.

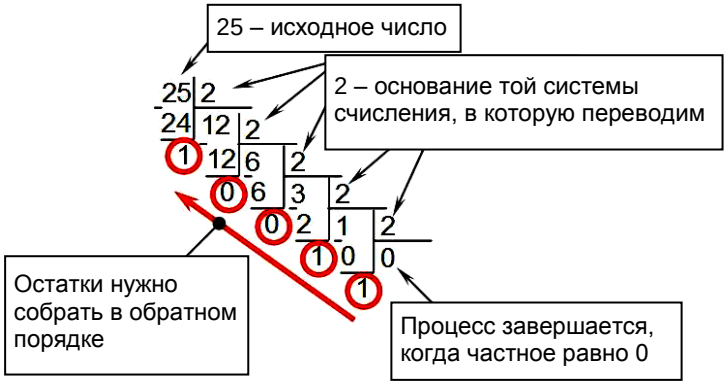


Рисунок 1.9 – **Перевод числа 25 из десятичной системы счисления в двоичную**

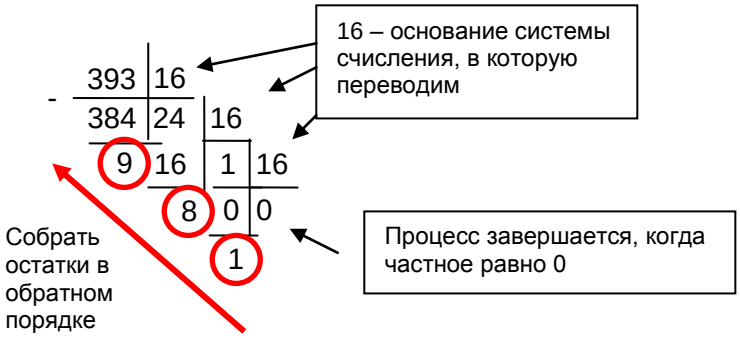


Рисунок 1.10 – **Перевод числа 393 из десятичной системы счисления в шестнадцатеричную**

1.3. Хранение чисел в памяти компьютера

Вся информация в памяти компьютера представляется в двоичной системе счисления. Один разряд двоичного числа называется *битом*. Бит может принимать значения 0 или 1. Однако отдельный бит памяти компьютера не имеет своего адреса, т. е. к нему нельзя обратиться как к самостоятельной единице.

Минимальная адресуемая единица памяти называется *байтом*. В большинстве компьютеров байт состоит из восьми бит. С помощью такого количества двоичных разрядов можно закодировать 256 различных целых чисел, а максимальное число, которое помещается в один байт, равно 255.

Почему именно так? Чтобы понять этот факт, представим, что у нас два бита. Сколько различных комбинаций из двух нулей и единиц можно придумать? Очевидно, 4. Все они (а также их десятичные эквиваленты) представлены в таблице 1.2. При этом $4 = 2^2 = 2^{\text{количество разрядов}}$.

Таблица 1.2 – **Возможные комбинации двоичных цифр в двух разрядах**

Двоичное число	Соответствующее десятичное число
00	0
01	1
10	2
11	3

Теперь представим, что у нас три бита. Возможные комбинации нулей и единиц показаны в таблице 1.3. Их всего 8, при этом $8 = 2^3 = 2^{\text{количество разрядов}}$.

Таким образом, легко заметить закономерность: если у нас n бит, то с их помощью можно закодировать 2^n различных двоичных чисел. Причем максимальное из этих чисел будет равно $2^n–1$ (оно представляет собой единицы во всех разрядах).

Таблица 1.3 – **Возможные комбинации двоичных цифр в трех разрядах**

Двоичное число	Соответствующее десятичное число
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Именно поэтому в байте можно разместить 256 различных целых чисел: $2^8 = 256$. Причем это числа от 0 до 255. Но это в том случае, если нужно хранить только неотрицательные числа.

Прежде чем говорить об отрицательных числах, следует рассмотреть правило выполнения операции сложения чисел в двоичной системе счисления. Операция сложения в двоичной системе выполняется поразрядно. При этом, если складываются две единицы, то получаем 10, т. е. единица переносится в старший разряд. Например, на рисунке 1.11 проиллюстрирован процесс определения суммы $11_2 + 1_2$.

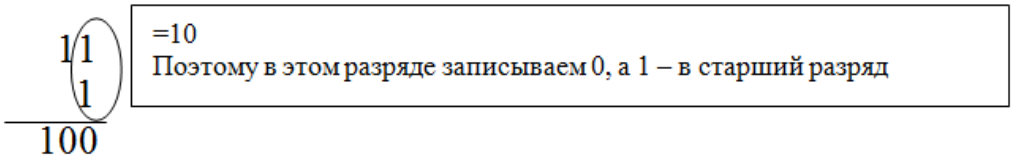


Рисунок 1.11 – **Определение суммы двоичных чисел**

Рассмотрим теперь принцип хранения отрицательных целых чисел. Для этого решили первый (старший) бит числа отводить для кодирования знака: 0 – число неотрицательное, 1 – отрицательное. Тогда для хранения собственно значения (модуля) числа остается на один бит меньше (в байте – 7 бит). А значит, максимальное положительное число, которое можно так закодировать, будет $2^7 - 1 = 127$ (это соответствует семи единицам).

Если же число отрицательное, то кроме наличия 1 в старшем (знаковом) разряде, само значение числа переводится в дополнительный код согласно следующему алгоритму:

1. Двоичное представление модуля числа инвертируется (т. е. 0 заменяется на 1, а 1 – на 0).
2. К полученному результату прибавляется 1.

Пример 1.5. Дано число -18_{10}. Нужно перевести его в двоичный код и разместить в одном байте. Алгоритм решения будет следующий:

1. Берем абсолютное значение (модуль) числа и переводим в двоичный код: $18_{10} = 10010_2$ (алгоритм перевода приведен выше).
2. Поскольку число нужно разместить в одном байте, дополним его слева незначащими нулями до 8 разрядов: 00010010.
3. Инвертируем двоичное представление числа: 11101101.
4. Прибавляем 1:

$$+\begin{array}{r} 11101101 \\ 1 \\ \hline 11101110 \end{array}.$$

В итоге получаем число 11101110. Именно так выглядит в памяти отрицательное число –18.

Вещественным называется число, которое может иметь дробную часть. Будем себе представлять именно десятичное дробное число, а не дробь вида $\frac{a}{b}$. Например, нужно разместить в

памяти число 0,058. Перед размещением в памяти такого числа компьютер его нормализует: представляет в виде $0{,}58 \cdot 10^{-1}$.

При этом число 0,58 называется *мантиссой* (первая цифра после запятой должна быть отлична от нуля), а число –1 именуется *порядком*. Кстати, в таком почти виде число можно вывести и на экран: 0.58E-1. Такая форма записи называется *экспоненциальной*, а форма 0,058 – *числом с плавающей точкой*.

В памяти компьютера ячейка, которая отводится под хранение вещественного числа, делится на две части: в одной хранится мантисса (со знаком), а в другой – порядок (со знаком) (рисунок 1.12). Количество бит, отводимое под них, определяется реализацией компилятора.

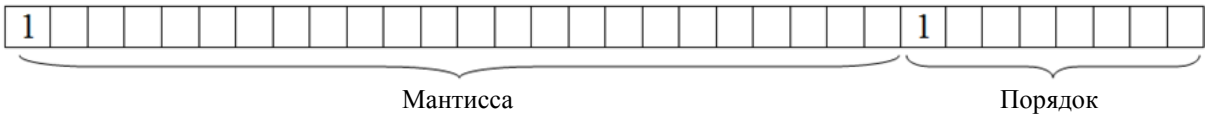


Рисунок 1.12 – **Принцип размещения вещественного числа в памяти компьютера**

Если же нужно данное число вывести на экран в привычном нам виде (с фиксированной точкой), то производится обратное преобразование: мантисса умножается на $10^{\text{порядок}}$.

1.4. Адресация данных в памяти компьютера

Минимальной адресуемой единицей памяти является один байт. Это означает, что для выполнения операции процессор может обратиться только к байту целиком, а не к какой-то его части.

Чтобы иметь доступ к байту памяти (или набору байтов), процессор должен знать его адрес. Адрес хранится в специальном регистре процессора. В 32-разрядных операционных системах под регистр адреса отводится 32 бита. Поэтому диапазон возможных адресов составляет $0 \ldots 2^{32} - 1$. Максимальный адрес будет: $2^{32} - 1 = 4\,294\,967\,295$ байта ≈ 4 Гбайт.

Поэтому, если на компьютере с 32-разрядной операционной системой установить микросхемы оперативной памяти с объемом более 4 Гбайт, то система просто «не увидит» эту память и не сможет с ней работать.

Можете подсчитать сами, какой объем оперативной памяти будет доступен для 64-разрядных операционных систем.

1.5. История развития языка С

Язык программирования С появился в 1972 г. Это один из примеров долгожительства среди языков программирования.

Он был задуман как инструментальный язык для разработки операционных систем и создавался одновременно с операционной системой UNIX. Авторами этого языка и ОС UNIX являются американские программисты Деннис Ритчи и Кеннет Томпсон, работавшие в компании «Bell Labs».

Язык назвали «Си» (С – третья буква английского алфавита), потому что многие его особенности берут начало от старого языка «Би» (В – вторая буква английского алфавита).

«Си» сочетает в себе черты как языка высокого уровня, так и машинно-ориентированного языка, допуская программиста ко всем машинным ресурсам, чего не обеспечивают такие языки, например, как Бейсик и Паскаль.

Чтобы лучше понять, какое место занимает язык С среди современных средств разработки программного обеспечения, нужно знать виды языков программирования (процедурные, аппликативные, языки системы правил, объектно-ориентированные).

Самые первые языки программирования, такие как Фортран, КОБОЛ, АЛГОЛ, создавались как *процедурные языки*. К этому типу относится и язык программирования С. Программа на таком языке – это последовательность операторов, которые изменяют значения ячеек памяти:

```
operator1; operator2; operator3;
```

Наиболее яркий представитель *аппликативных* (или функциональных) языков программирования – язык Лисп. Программа на таком языке – это набор функций, которые последовательно применяются к начальным данным:

```
function1(function2(function3(beginning_date)));
```

Языки системы правил (языки логического программирования) описывают условия, при выполнении которых могут происходить те или иные действия. К этому виду относится язык Prolog. Код программы выглядит приблизительно так:

```
if condition1 then operator1;
if condition2 then operator2;
if condition3 then operator3;
```

Объектно-ориентированные языки получили в настоящее время наибольшее распространение в промышленном программировании. Они основаны на понятии объекта как объединения данных и операций над ними. К таким языкам относятся, например, C++, Java, C# и Pithon. Эти языки фактически объединяют и расширяют идеи процедурного и функционального программирования.

Таким образом, при объектно-ориентированном программировании отдельные фрагменты кода все равно разрабатываются как процедурные алгоритмы. Поэтому обучение программированию по специальности «Экономика электронного бизнеса» начинается с изучения языка С, который даст возможность развить алгоритмическое мышление и усвоить основные приемы работы с данными. При этом он обеспечит легкий переход к объектно-ориентированному языку Java, поскольку их синтаксис очень похож.

Первое описание языка С было дано в книге Б. Кернигана и Д. Ритчи «Язык программирования С» (которая была переведена и на русский язык). Долгое время это описание являлось стандартом, однако ряд моментов допускал и неоднозначное толкование, которое породило множество трактовок языка С. Для исправления этой ситуации при Американском национальном институте стандартов (ANSI) был образован комитет по стандартизации языка С. В 1989 г. был утвержден стандарт языка С, получивший название ANSI C. В Европе этот стандарт был фактически переутвержден с именем ISO C.

В начале 80-х гг. Бьерном Страуструпом в результате дополнения и расширения языка С был создан новый по сути язык, получивший название «С с классами». В 1983 г. это название было заменено на «C++».

С этих пор языки С и C++ развивались параллельно. Оставаясь языком процедурного программирования, язык С приобретал все больше черт и возможностей языка C++. Это привело к появлению новых стандартов: C99 (1999 г.) и C11 (2011 г.). Некоторые компиляторы не поддерживают эти новые стандарты, оставаясь в пределах ANSI C.

1.6. Интегрированная среда разработки

Язык С является языком высокого уровня. Это означает, что программа пишется на языке, приближенном к естественному человеческому языку, а затем автоматически переводится на язык машинных кодов. Процесс перевода с языка высокого уровня на машинный язык называется *трансляцией.* Трансляция бывает двух видов (интерпретация и компиляция).

Интерпретация заключается в том, что преобразование кода происходит в процессе выполнения программы. Интерпретатор берет очередной оператор программы, переводит его в машинный код и сразу же выполняет. Затем берет следующий оператор и т. д. (рисунок 1.13). То есть результат преобразования (машинный код) нигде не сохраняется. Минус этого подхода заключается в том, что скорость выполнения программы не может быть очень высокой.

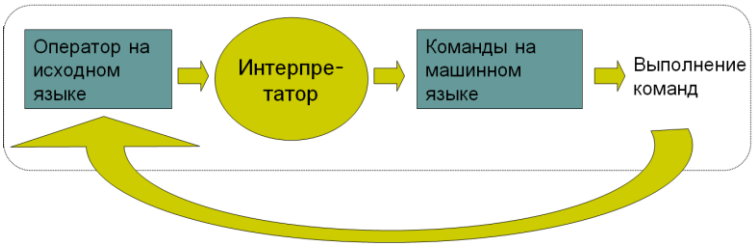


Рисунок 1.13 – **Принцип работы интерпретатора**

В случае *компиляции* программа – компилятор – сразу переводит весь код программы в машинный код. В результате получается файл с расширением .obj, который можно сохранить на диске. Затем этот файл обрабатывает линковщик (компоновщик), который добавляет к коду самой программы функции из библиотеки (например, функции ввода – вывода) и настраивает связи между ними. В результате получается загрузочный модуль (с расширением .exe), который также можно сохранить на диск. При запуске такого файла он выполняется автономно, без участия исходной программы и компилятора (рисунок 1.14).

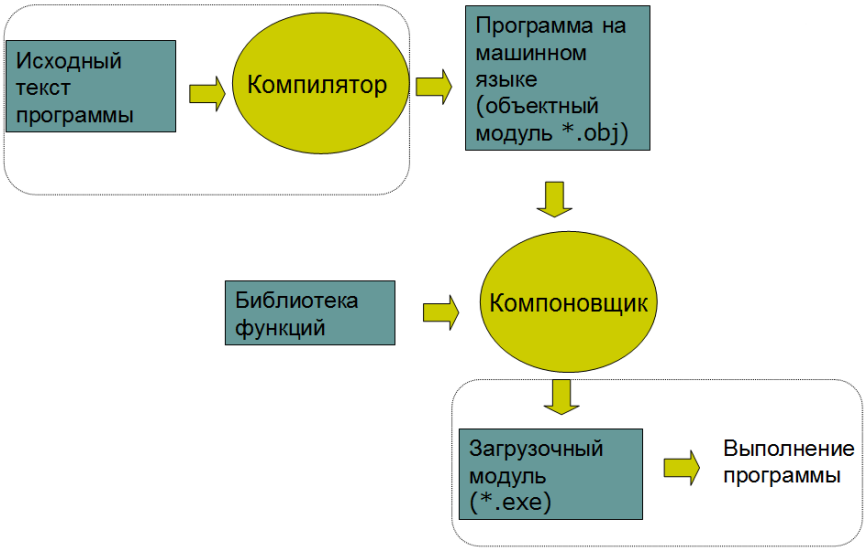


Рисунок 1.14 – **Принцип работы компилятора и компоновщика**

Язык С является языком компилируемого типа, поэтому для работы с программой на этом языке нужно иметь компилятор и компоновщик.

Интегрированная среда разработки (Integrated Development Environment, IDE) – это специальное приложение, которое позволяет упростить разработку программ на языке высокого уровня. Она обычно включает в себя:

- текстовый редактор (чтобы набрать текст программы, сохранить его на диске, редактировать и т. д.);
- компилятор и компоновщик (чтобы перевести программу в машинный код);
- средства отладки и запуска программ (чтобы обеспечить удобства поиска ошибок);
- стандартные библиотеки, содержащие многократно используемые элементы программ;
- справочную систему и др.

Использование среды разработки не обязательно. Можно набрать текст программы в обычном текстовом редакторе (например, Блокноте), а для компиляции использовать отдельный компилятор языка. Но работать со средой разработки гораздо удобнее и комфортнее.

Для языка С существует достаточно много различных сред разработки. В данном пособии описано использование среды Microsoft Visual Studio. На сегодняшний день доступна бесплатная версия Visual Studio Community 2017. Она мультиязыковая и подходит для .NET-совместимых языков (C++, C# и VB.NET). То есть фактически студенты работают с компилятором C++, но используют подмножество этого языка, которое соответствует языку С.

Установить Microsoft Visual Studio можно с официального сайта www.visualstudio.com. Во время установки программы на компьютер Интернет должен быть доступен. Рекомендуется

устанавливать англоязычную версию программы (в учебных классах установлена русскоязычная).

Если ресурсов персонального компьютера не хватает для того, чтобы установить эту программу (см. системные требования на сайте), то можно установить более раннюю версию Visual Studio Express 2010 (https://www.microsoft.com/ru-ru/SoftMicrosoft/VisualStudioExpress.aspx). Эта версия представляет собой набор продуктов для каждого языка отдельно. Необходимо установить Visual C++ 2010 Express.

Через некоторое время после установки сайт затребует ключ. Нужно продолжить регистрацию и получить ключ бесплатно.

Далее описан принцип работы со средой Visual Studio Community 2017 (русскоязычная версия).

1.7. Приемы работы в IDE Microsoft Visual Studio

После запуска программы появляется окно с рабочей областью, поверх которой открыто окно «Начальная страница» (рисунок 1.15).

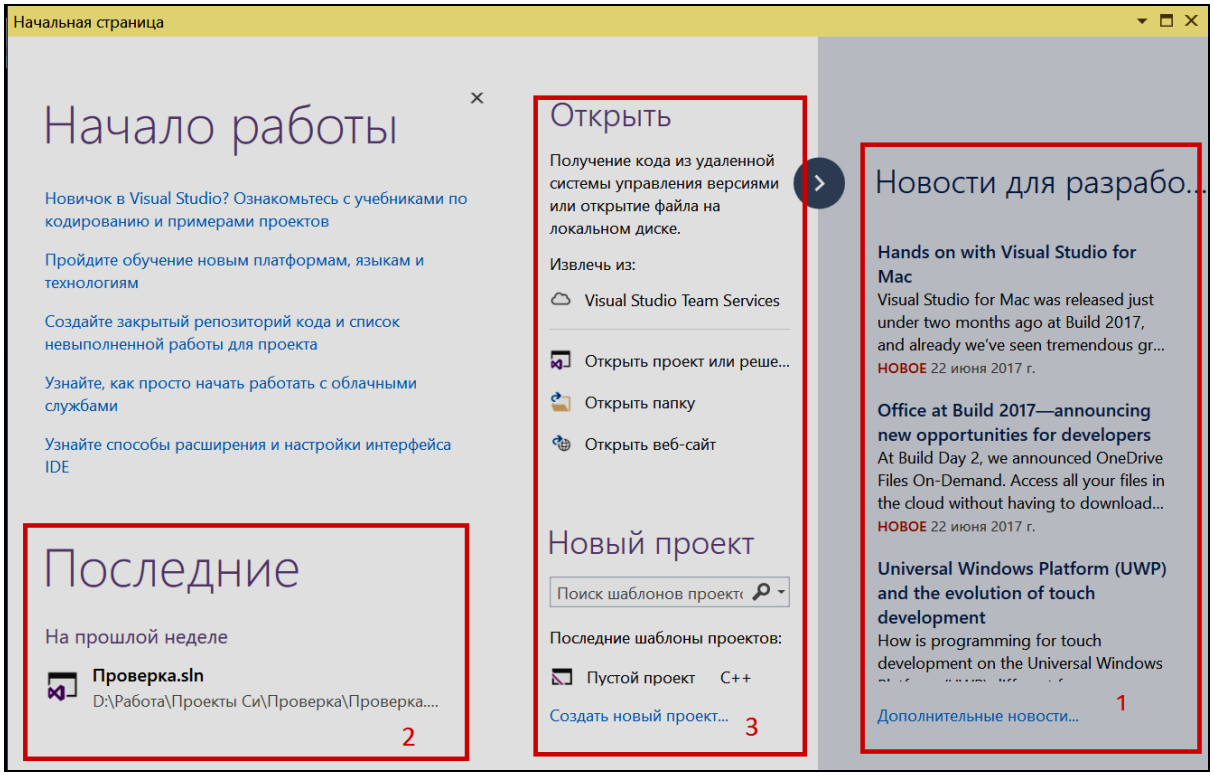


Рисунок 1.15 – **Начальная страница среды разработки Visual Studio**

Эта страница содержит зоны:

1) с различными новостями от разработчиков приложения;

2) список проектов, с которыми недавно работали (удобно, если хотите открыть один из них и продолжить работу);

3) команды создания нового проекта и открытия ранее созданного с возможностью выбора их на диске.

Рассмотрим общий способ создания проекта без использования начальной страницы.

Создание проекта

В главном меню выберите *Файл* → *Создать* → *Проект*. Появится окно «Создание проекта (рисунок 1.16), в котором нужно выбрать шаблон *Visual C++* (1), вид проекта *Пустой проект* (2), задать имя проекта, например *Hello* (3).

Имя решения автоматически задается такое же, как и имя проекта. Но его можно изменить. Например, на рассматриваемом рисунке задано имя решения *Lesson 1*. В поле «Расположение» нужно задать путь, по которому будут записаны файлы решения. (Обратите внимание, какой путь задаете, чтобы потом не пришлось разыскивать свои файлы по всему компьютеру.)

Флажок *Создать каталог для решения* (4) убирать не рекомендуется.

В результате будет создан проект с именем *Hello* в решении *Lesson 1*, что отражено в окне обозревателя решений в виде соответствующих папок (рисунок 1.17).

Если в рабочей области программы не появилось окно обозревателя решений, то его можно вывести на экран, выбрав в меню *Вид* (рисунок 1.18). Аналогично можно отобразить и все прочие окна программы.

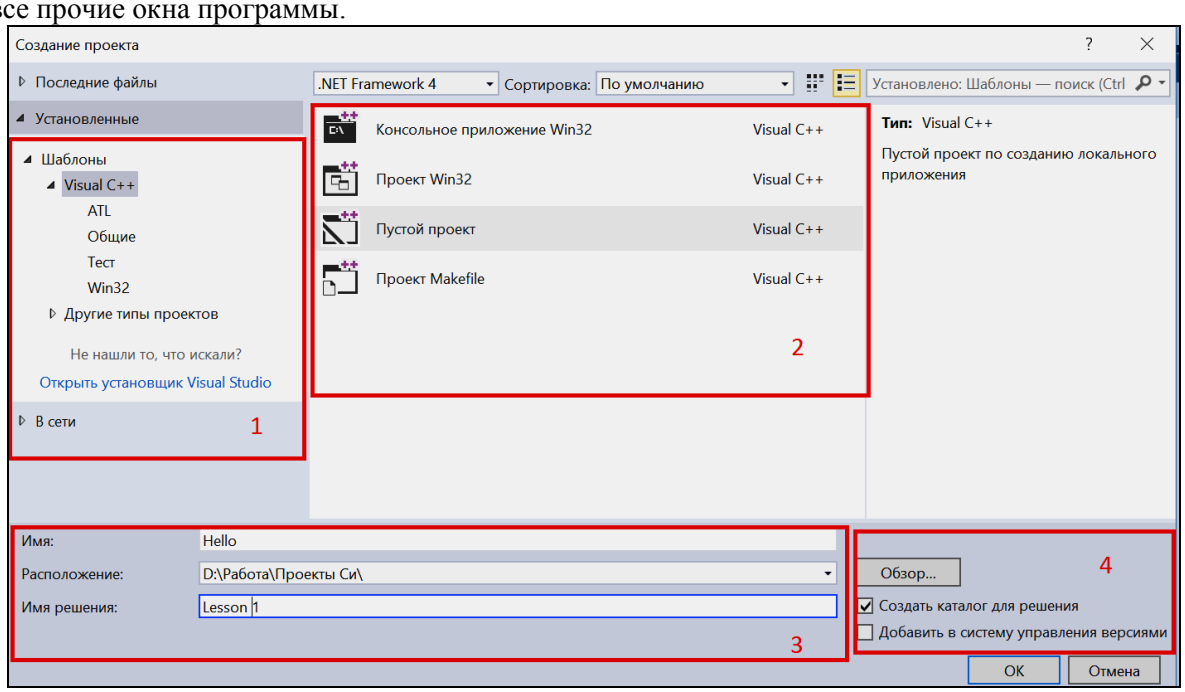


Рисунок 1.16 – **Окно создания проекта**

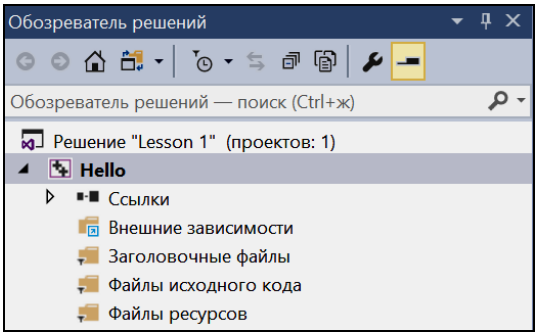


Рисунок 1.17 – **Обозреватель решений после создания нового проекта**

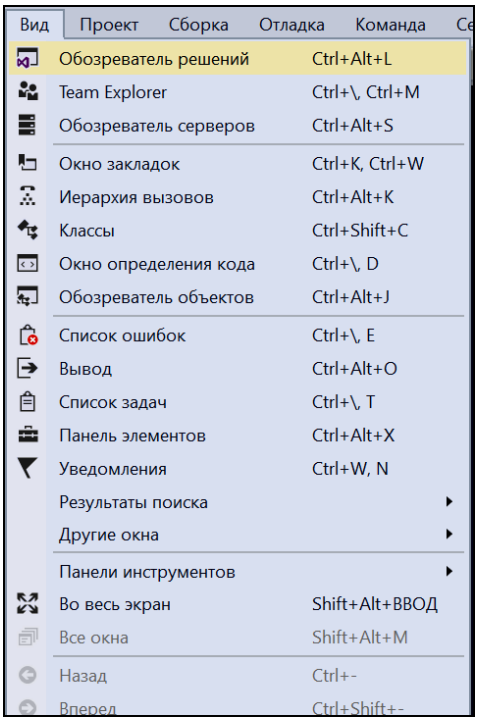


Рисунок 1.18 – **Вывод на экран различных окон приложения Visual Studio**

Далее в проект нужно добавить файл программы на языке С. Для добавления файла выберем в окне «Обозреватель решений» папку *Файлы исходного кода* и правой кнопкой мыши вызовем контекстное меню. В меню нужно выбрать *Добавить* → *Создать элемент*.

Появится окно выбора типа файла (рисунок 1.19). В этом окне выбираем *Файл C++(.cpp)*, задаем имя файла (может совпадать с именем проекта). Папку расположения файла лучше оставить по умолчанию. После задания параметров файла нажимаем *Добавить*.

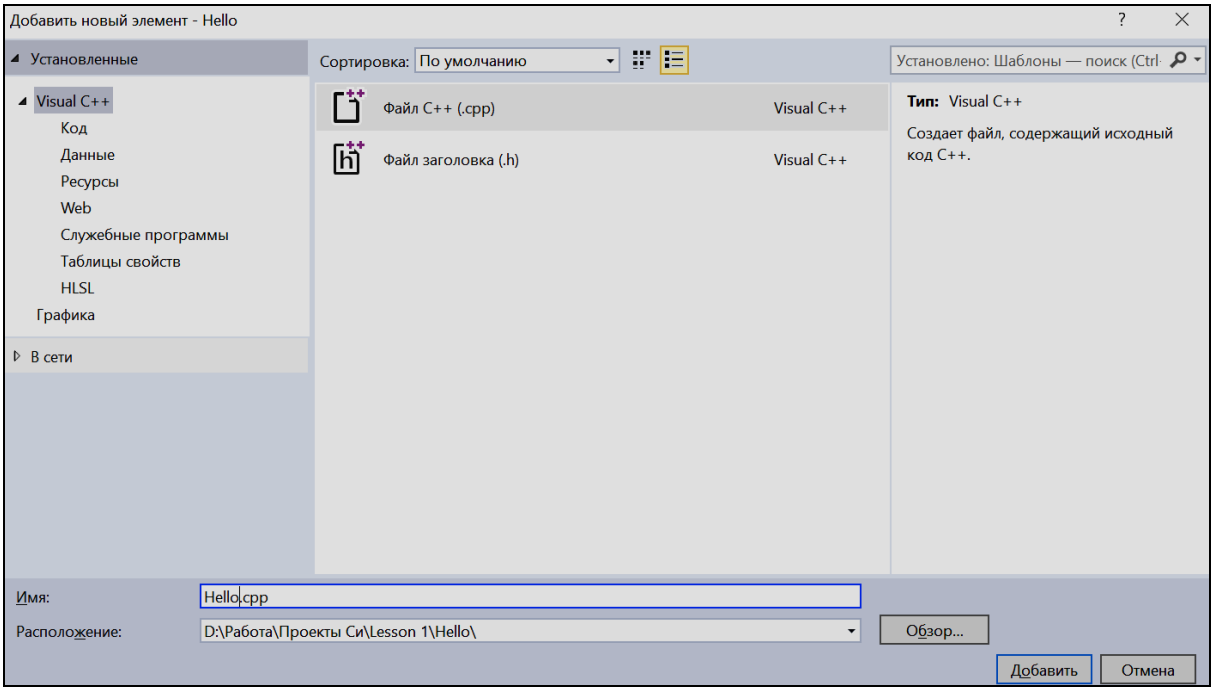


Рисунок 1.19 – **Задание параметров файла программы**

После создания файла он отображается в папке проекта в окне «Обозреватель решений». Кроме того, автоматически открывается окно, в которое можно ввести текст программы (рисунок 1.20). Смысл операторов этой программы будет рассмотрен позже. Обычно первая программа, которую создают на любом языке программирования, выводит на экран слова «Hello, world!».

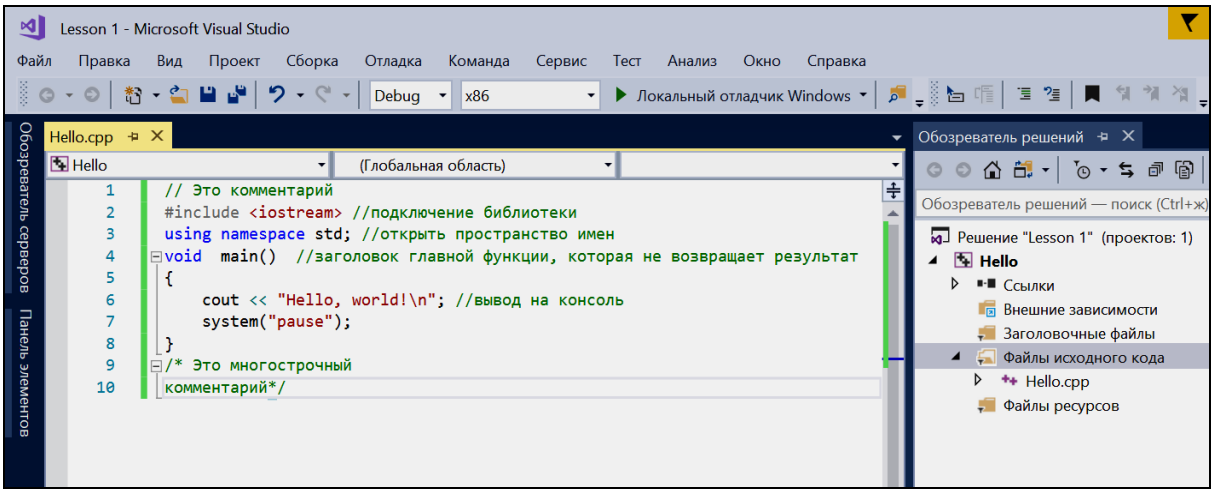


Рисунок 1.20 – **Текст программы, созданный встроенным текстовым редактором**

Обратите внимание, что встроенный текстовый редактор Visual Studio автоматически выделяет разным цветом различные объекты текста программы: ключевые (служебные) слова – синим, строковые литералы (в двойных кавычках) – красным, а комментарии – зеленым.

Если на ярлычке окна с текстом программы имеется звездочка, это означает, что после изменений текста он еще не был сохранен на диске. Можно задать *Файл* → *Сохранить…* или *Файл* → *Сохранить все* (в этом случае сохраняется не только текущий файл, но и все файлы решения). Кроме того, код автоматически сохраняется на диске при компиляции или запуске проекта на выполнение.

Компиляция, компоновка и выполнение программы

Для выполнения компиляции и компоновки проекта нужно выбрать *Сборка* → *Собрать решение* (рисунок 1.21). Альтернативный вариант – нажать клавиши Ctrl + Shift + B.

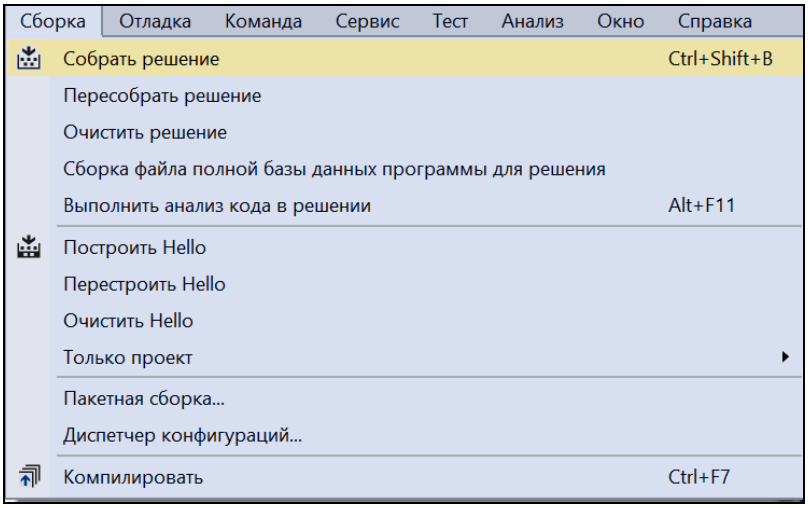


Рисунок 1.21 – **Сборка проекта**

Если в программе имеются синтаксические ошибки, то система выведет соответствующее предупреждение, а список ошибок будет выведен в окне «Список ошибок» (рисунок 1.22). Двойной щелчок мышью на ошибке установит курсор на то место кода, где (по мнению компилятора) находится ошибка. При этом на самом деле ошибка может быть не только в данном операторе, но и в предыдущем (и даже в другом месте кода).

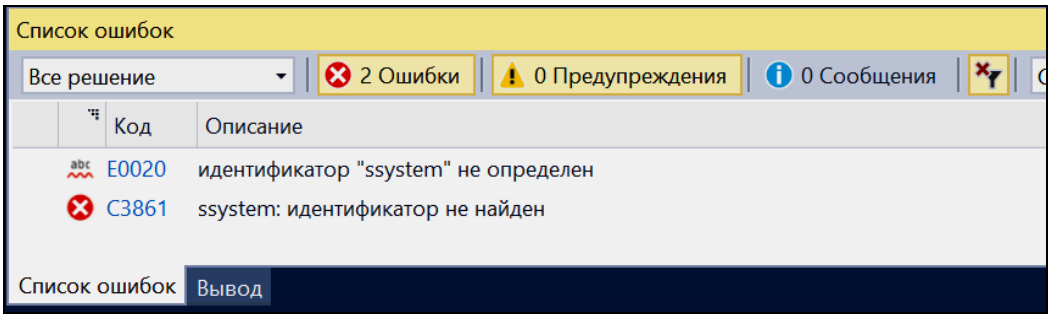


Рисунок 1.22 – **Окно «Список ошибок»**

Если же ошибок не было, и компиляция прошла успешно, можно запустить программу на выполнение, задав команду *Отладка* → *Начать отладку* (клавиша F5). Также на панели инструментов имеется кнопка в виде зеленого треугольника. Нажатие этой кнопки позволяет выполнить последовательно сразу все этапы: компиляцию, компоновку (сборку) и запуск на выполнение.

Результатом выполнения программы будет появление на экране компьютера окна, где на черном фоне белыми буквами будет выведено *Hello, world!* При этом программа будет ожидать нажатия любой клавиши, а затем окно исчезнет с экрана компьютера (рисунок 1.23).

Это окно и называется консолью для вывода. Ввод с консоли привязан к клавиатуре. Будем рассматривать пока только консольный ввод – вывод, а в дальнейшем изучим другие способы организации ввода – вывода на экран, например, работу с формами.

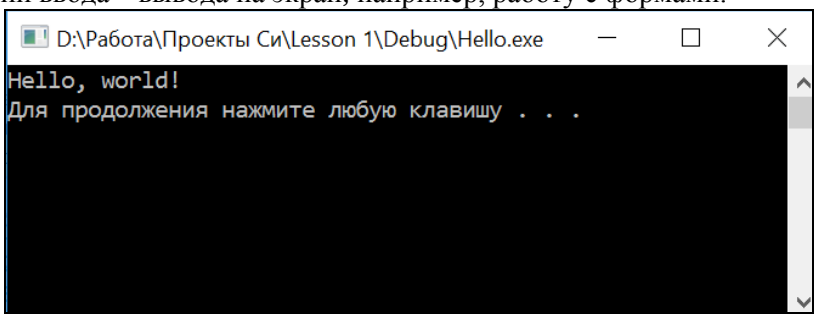


Рисунок 1.23 – **Окно консоли**

Если программа уже была откомпилирована, а затем внесли какие-то изменения и запустили ее на выполнение вновь, то система спросит, следует ли выполнить повторную компиляцию (рисунок 1.24). Разумеется, следует согласиться, чтобы изменения были учтены в программе.

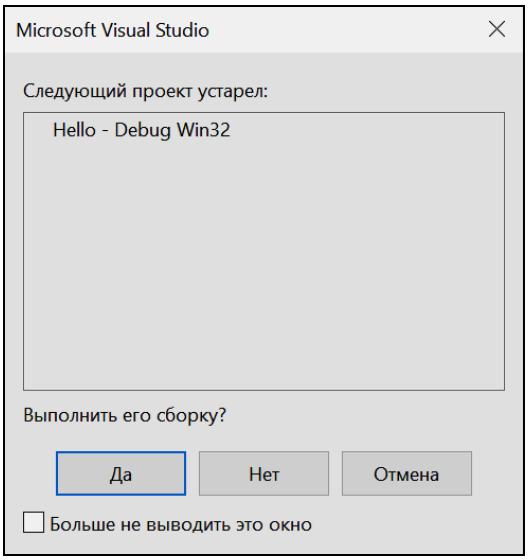


Рисунок 1.24 – **Запрос на повторную сборку проекта**

Понятие проекта и решения

Проект (project) – это набор файлов, которые затем будут скомпилированы и скомпонованы в один исполняемый файл (.exe).

Решение, или *сборка (solution)*, – совокупность проектов, которые по каким-то причинам нужно объединить и хранить вместе. Например, каждая задача модуля оформляется как проект, а все проекты одного модуля объединяются в решение. Один из проектов можно назначить запускаемым (активным), выбрав соответствующий пункт в его контекстном меню. Тогда именно этот проект будет компилироваться, компоноваться и запускаться на выполнение, когда выбирается соответствующая команда меню. В окне обозревателя решений имя запускаемого проекта выделяется жирным шрифтом.

Теперь рассмотрим пример создания второго проекта в решении. Как уже известно, при создании первого проекта решение создается автоматически.

Выбираем в окне «Обозреватель решений» решение *Lesson 1* и вызываем контекстное меню (правой кнопкой мыши), а в нем *Добавить* → *Создать проект…*.

Появится окно для добавления нового проекта, которое уже ранее было описано. При этом имя решения будет отсутствовать, так как проект помещается уже в существующее решение. Далее новый проект появится в окне обозревателя решений (рисунок 1.25). Чтобы с ним работать, нужно назначить его запускаемым проектом. Для этого следует щелкнуть по имени проекта в обозревателе решений правой кнопкой мыши и выбрать из контекстного меню *Назначить автозагружаемым проектом*.

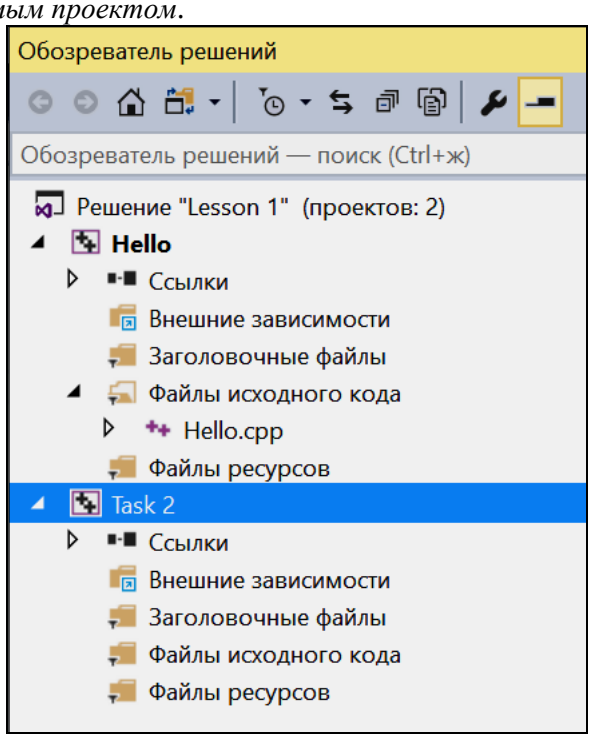


Рисунок 1.25 – **Решение, которое состоит из двух проектов**

Для каждого проекта создаются следующие папки:

• Файлы исходного кода (Source Files) – папка, которая содержит разные части программы на языке С (с расширением .cpp). Пока не будем разбивать код программы на части, поэтому в этой папке будет только один файл. В этом файле обязательно должна быть функция main(), с которой начинается процесс выполнения программы.

• Файлы ресурсов (Resource Files). Самый распространенный пример ресурса – это различные изображения.

• Внешние зависимости (External Dependencies). Эта папка содержит ссылки на стандартные библиотеки и заполняется средой разработки.

• Заголовочные файлы (Header Files) – файлы с расширением .h, которые создает программист. Заголовочные файлы будем подробно разбирать позже, когда перейдем к распределению кода программы на несколько файлов.

1.8. Первая программа

Рассмотрим текст первой программы (рисунок 1.26), которая выводит приветствие на консоль.

```
//Это комментарий
#include <iostream>
using namespace std; //открыть пространство имен
void main()       // заголовок главной функции, которая не возвращает результат
{
        cout<<"Hello, world!\n";     //вывод на консоль
        system( "pause" );           //задержка выполнения программы
}
/* Это многострочный
комментарий*/
```

Рисунок 1.26 – **Текст первой программы**

То, что компилятор выделил зеленым цветом, является комментарием. Комментарии служат для пояснения фрагментов программы и могут быть записаны на любом языке. Компилятор комментарии не обрабатывает. Однострочный комментарий начинается с двух символов «//», а многострочный заключается в скобки «/*…*/».

Первая строка программы #include<iostream> является директивой препроцессора, т. е. эта команда обрабатывается до начала компиляции всей программы. Суть этой конкретной директивы заключается в том, что вместо нее подставляется (подключается) заголовочный файл стандартной библиотеки iostream, в которой, в частности, описаны средства ввода – вывода. Некоторые библиотеки в Visual Studio подключаются по умолчанию, поэтому для них нет необходимости записывать директиву #include. Но для того чтобы код программы был переносимым и мог обрабатываться другими компиляторами (а не только Visual Studio), лучше всегда явно подключать стандартные библиотеки.

В библиотеках все имена функций сгруппированы в пространства имен. Это сделано для того, чтобы предотвратить конфликты между именами, используемыми в библиотеках, и именами, которые дает программист. Библиотеке iostream соответствует пространство имен std. Полное имя из библиотеки состоит из двух частей: имени пространства имен и (через два двоеточия) собственно идентификатора функции или другого объекта, например, std::cout.

Чтобы упростить использование библиотечных имен, следует «открыть» пространство имен, задав оператор using namespace std. После этого можно использовать только вторую часть имени (cout), а компилятор будет по умолчанию полагать, что это имя из пространства std. Однако открывая пространство имен, программист берет на себя ответственность за отсутствие конфликта между его собственными именами и именами из библиотеки (например, теперь нельзя назвать свою переменную cout).

Следующая строка – это заголовок функции, которая называется main. Функция с именем main – это особая функция, с которой начинается выполнение программы. Понятно, что такая функция должна быть в проекте только одна. На то, что это функция, указывают две скобки, внутри которых могут быть параметры. Но в нашем примере функция main используется без параметров.

Слово «void» перед именем функции означает тип результата, который функция возвращает; void – это пустой тип (обозначает, что функция не возвращает никакого результата).

Фигурные скобки означают начало и конец блока, т. е. последовательность операторов, которая рассматривается как единое целое. В данном случае это начало и конец тела функции main.

Следующая строка кода – это оператор вывода на экран компьютера текста «Hello, world!». Слово «cout» (сокращение от console out) означает выходной поток символов, связанных с консолью. Оператор вставки, отображаемый как <<, позволяет программе вставлять символы в выходной поток, т. е. фактически написано, что строка «Hello, world!» направляется в выходной поток консоли. Потоковый ввод – вывод является элементом языка C++. Позже будет рассмотрен классический ввод – вывод языка C с использованием функций printf() и scanf().

Сама выводимая строка заключена в двойные кавычки. Внутри этой строки имеется особый символ '\n'. Он означает переход к новой строке. Это действительно один символ, а не два, так как в памяти занимает один байт, как все символы. Символы, которые начинаются с косой черты, называются *escape-последовательностями*.

И наконец, последняя строка (system("pause");) написана для того, чтобы экран консоли не пропадал после того, как выполнен вывод. Это вызов функции, которая выводит на консоль сообщение «Для продолжения нажмите любую клавишу» и приостанавливает выполнение программы до тех пор, пока пользователь не нажмет любую клавишу на клавиатуре.

Следует отметить, что при использовании русских букв может возникнуть проблема. Дело в том, что среда разработки Visual Studio использует кодовую таблицу Windows, а в консольном окне по умолчанию – кодировка MS DOS. То есть один и тот же код в этих кодовых таблицах соответствует разным символам. Чтобы использовать русские буквы, нужно вызвать функцию `setlocale (LC_ALL, "Russian")` или `setlocale (LC_ALL, "rus")`, которая установит русскую локаль.

Русификация консольного окна будет работать, если в операционной системе Windows в настройках языков и стандартов была установлена кириллица по умолчанию.

1.9. Основные понятия языка

После знакомства с первой программой и принципами работы в среде Visual Studio перейдем к планомерному изучению синтаксиса и семантики языка программирования C. В описании любого языка можно выделить четыре основных элемента: символы, слова, словосочетания и предложения. Подобные элементы содержит и язык программирования, только слова здесь называют лексемами (элементарными конструкциями), словосочетания – выражениями, а предложения – операторами.

Алфавит языка C включает:

- прописные и строчные латинские буквы и символ подчеркивания, который употребляется наряду с буквами;
- арабские цифры от 0 до 9;
- специальные символы, например +, *, { и &;
- пробельные символы: пробел, символы табуляции, символы перевода строки и формата.

Алфавит языка в стандарте называется базовым набором символов. Кроме того, существует понятие «набор символов реализации» – все множество символов, доступных на данном компьютере. Он содержит базовый набор в качестве подмножества.

Обратите внимание, что символы национальных алфавитов (в частности, русские) не входят в базовый набор символов по стандарту. Именно поэтому не рекомендуется использовать русские буквы при задании имен в программе (хотя Visual Studio это позволяет).

Из символов базового набора составляются лексемы языка и директивы препроцессора. Символы из набора реализации используются для написания комментариев.

Лексемы языка программирования аналогичны словам обычного языка. Это минимальные единицы языка, которые компилятор отличает и обрабатывает как единое целое. Например, лексемами являются константа 128 (но не ее часть 12), имя `Vasia`, ключевое слово `goto` и знак операции сложения «+». Из лексем составляются выражения и операторы.

Различают следующие виды лексем:

- имена (идентификаторы);
- ключевые слова;
- знаки операций;
- разделители;
- литералы (константы).

Выражение задает правило вычисления некоторого значения. Например, выражение *a* + *b*, задает правило вычисления суммы величин *a* и *b*.

Оператор задает законченное описание некоторого действия.

Операторы делят на исполняемые и неисполняемые, простые и составные. Исполняемые операторы задают действия над данными. Неисполняемые операторы служат для описания данных, поэтому их часто называют операторами описания или просто описаниями. Например, int a; – это оператор описания целочисленной переменной *a*. Каждый оператор должен завершаться точкой с запятой (;).

Составной оператор, или блок, – это группа операторов, заключенная в фигурные скобки. Блоки могут быть вложенными.

Рассмотрим подробнее разные виды лексем.

Идентификатор – это имя программного объекта (переменной, функции, константы и т. д).

Правила составления идентификаторов следующие:

- В идентификаторе могут использоваться латинские буквы, цифры и знак подчеркивания.
- Прописные и строчные буквы различаются.
- Первым символом идентификатора может быть буква или знак подчеркивания (но лучше начинать идентификатор с буквы, так как со знака подчеркивания обычно начинаются служебные идентификаторы).
- Идентификатор не должен совпадать с ключевыми словами и именами используемых стандартных объектов языка.
- Длина идентификатора по стандарту не ограничена.

Следует придумывать подходящие имена. Имя должно отражать смысл хранимой величины, отвечать принятой нотации (правилам составления имен), быть легко распознаваемым и, желательно, не содержать символов, которые можно перепутать друг с другом (например, 1, *l* и *i*).

Ключевые слова – это зарезервированные идентификаторы, которые имеют специальное значение для компилятора. Их можно использовать только в том смысле, в котором они определены. Например, «do», «int», «void» и т. д. В Visual Studio они выделяются синим цветом.

Знак операции – это один или более символов, определяющих действие над операндами. Внутри знака операции пробелы не допускаются. Символы, составляющие знак операции, могут быть как специальными (&&, | и <), так и буквенными (new, sizeof).

Литералы – это фиксированные значения (константы), которые не имеют имени. Но при этом литералы имеют тип, который компилятор определяет по их внешнему виду. Например, если в программе записано число 456, то это литерал целого типа, если число 45.6, то это литерал вещественного типа. Подробнее литералы будут рассмотрены после изучения типов данных. Пока остановимся только на строковом литерале – наборе символов в двойных кавычках.

Внутри строковых литералов можно использовать специальные управляющие символы, называемые *escape-последовательностями*. Они представляют собой комбинацию символа «\» и символа, определяющего действие, которое необходимо произвести над строкой. Примеры escape-последовательностей приведены в таблице 1.4.

Таблица 1.4 – **Основные escape-последовательности**

Символ	Операция
\n	Перейти на начало новой строки
\t	Перейти к следующей позиции табуляции
\b	Удалить последний выведенный символ
\a	Подать звуковой сигнал
\\	Вывести обратную черту (\)
\"	Вывести двойную кавычку (")
\'	Вывести одинарную кавычку (')

Длинную строковую константу можно разместить на нескольких строках программы, используя в качестве знака переноса обратную косую черту, за которой следует перевод строки. Эти символы игнорируются компилятором, при этом следующая строка программы воспринимается как продолжение предыдущей. Например, строка

```
"Никто не доволен своей  \
```

```
внешностью, но каждый доволен  \
своим умом"
```

эквивалентна строке

```
"Никто не доволен своей внешностью, но каждый доволен своим умом".
```

Кавычки пишутся только в начале и в конце многострочного литерала.

Пример 1.6. Использовать escape-последовательности для форматирования строк, вывести на экран текст вида:

"To be or not to be…"
\Shakespeare\

Поскольку кавычки имеют служебный смысл (начало и конец строкового литерала), то для задания кавычки как символа внутри строки нужно использовать комбинацию \". Аналогично одна косая черта означает начало escape-последовательности, а две черты – просто один символ '\', который будет выведен на экран. Для перехода на новую строку будем использовать \n, а для шага табуляции – \t. В итоге получаем следующий оператор вывода на консоль:

```
cout<<"\"To be or not to be...\"\n\t\\Shakesreare\\\n";
```

Весь текст программы с использованием escape-последовательности будет следующий:

```
#include <iostream>
using namespace std;
void main()
{
    cout<<"\"To be or not to be...\"\n\t\\Shakesreare\\\n";
    system("pause");
}
```

В стандарте C11 определены *raw-строки* («сырые строки»). Это вид строковых литералов, в которых не обрабатываются escape-последовательности. Такие строки используются обычно, если нужно выводить на экран код на языках HTML, XML и других, в которых много служебных символов. Формат raw-строк: R"(строка без спецсимволов)".

Например, программа вывода цитаты из произведения В. Шекспира с помощью raw-строк выглядит так:

```
#include <iostream>
using namespace std;
void main()
{
    cout<<R"("To be or not to be..."     \Shakesreare\ )";
    system("pause");
}
```

Задачи

Задача 1.1. Описать алгоритм нахождения минимального из пяти чисел (с помощью блок-схемы и псевдокода).

Задача 1.2. Описать алгоритм определения степени числа (степень целая и положительная).

Задача 1.3. Описать алгоритм подсчета количества отрицательных чисел среди всех чисел, вводимых пользователем с клавиатуры. Числа вводятся до тех пор, пока пользователь не наберет число 0. После этого на экран выводятся результаты подсчета: общее количество введенных чисел и число отрицательных чисел среди них.

Задача 1.4. Преобразовать в десятичную систему счисления:
- 1001101_2;
- 0011010_2;
- 307_8;
- 307_{16};
- $3AF_{16}$.

Задача 1.5. Преобразовать в двоичную систему счисления из десятичной (предположить хранение в одном байте):
- 35;
- 242;
- –56;
- –21.

Задача 1.6. Преобразовать в шестнадцатеричную систему счисления из десятичной:
- 100;
- 526.

Задача 1.7. Написать программу, которая выводит на экран строку:
I study programming language "C/C++".

Задача 1.8. Написать программу, которая выводит на экран текст:
"Hello, World!"
\Dennis Ritchi\
Программа должна использовать однострочные комментарии.

Для самостоятельного решения

Задача 1.9. Описать алгоритм определения скидки на покупку. Если стоимость покупки больше 500 тыс. р., то скидка составляет 10%. Если же стоимость покупки равна от 250 до 500 тыс. р., то скидка – 5%. При стоимости покупки ниже 250 тыс. р. скидка не предусмотрена.

Задача 1.10. Описать алгоритм определения степени числа (степень целая и может быть как положительной, так и отрицательной).

Задача 1.11. Написать программу, которая выводит на экран строфу любимого стихотворения (или куплет песни).

Задача 1.12. Написать программу, которая выводит на консоль следующий текст:

```
#include <iostream>
using namespace std;
void main()
{
    cout<<"Privet, mir!";
}
```

Реализуйте эту задачу двумя способами: без использования raw-строк и используя это средство языка (если позволяет версия компилятора).

ТЕМА 2. ПЕРЕМЕННЫЕ И ТИПЫ ДАННЫХ

2.1. Типы данных языка С

Переменная – это именованная область памяти, предназначенная для хранения данных, которые могут быть изменены. Аналогично можно ввести понятие *«именованная константа»* –

область памяти, предназначенная для хранения постоянных данных (значение константы нельзя изменять в тексте программы, а переменной – можно). Каждая переменная или константа имеет имя и тип данных.

Тип данных определяет, сколько памяти выделяется компилятором для переменной или константы и как интерпретируется выделенная память. Как следствие, от типа зависит возможный диапазон значений переменной и то, какие операции над ней разрешены.

Типы языка С делятся на базовые и составные. *Базовые типы данных* являются неделимыми и позволяют описывать целые (целочисленные), вещественные, символьные и логические величины. На основе этих типов программист может конструировать составные типы. К *составным типам* относятся массивы, структуры, объединения, перечисления, ссылки, указатели.

В данной теме рассмотрим только базовые типы.

Целочисленные типы предназначены для хранения целых чисел. Такая переменная подойдет, например, для хранения счетчика количества цифр в числе или для номера дня недели, количества выигранных матчей за сезон и т. п.

В таблице 2.1 приведен список всех целочисленных типов с указанием соответствующих диапазонов значений. Отличаются целочисленные типы размером (количеством байт памяти, выделяемых для переменной) и тем, используется ли знаковый разряд для представления числа (т. е. можно ли хранить в такой переменной число со знаком или же только неотрицательные числа). Для явного указания того, что число является беззнаковым, служит модификатор unsigned. По умолчанию все целые типы, кроме char, считаются типами со знаком.

Почему именно такой диапазон значений соответствует типу, легко понять, если вспомнить представление целых чисел в памяти компьютера.

Таблица 2.1 – **Целочисленные типы языка C++**

Тип данных	Выделяемое количество байт памяти	Диапазон значений
char	1	От 0 до 255
signed char	1	От –128 до 127
short	2	От –32 768 до 32 767
unsigned short	2	От 0 до 65 535
int	4	От –2 147 483 648 до 2 147 483 647
unsigned int	4	От 0 до 4 294 967 296
long	4	От –2 147 483 648 до 2 147 483 647
unsigned long	4	От 0 до 4 294 967 296
long long	8	От –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
unsigned long long	8	От 0 до 2^{64}–1

При хранении целого числа со знаком в n битах старший разряд интерпретируется как знаковый. На представление собственно числа остается n – 1 бит. Поэтому диапазон значений такого числа от -2^{n-1} до $2^{n-1}-1$.

Если же число беззнаковое, то все n разрядов значимые и диапазон возможных значений – от 0 до 2^n-1.

Тип char предназначен для хранения кодов символов. В языке С эти коды символов могут также рассматриваться как целые числа и участвовать в арифметических операциях. По умолчанию тип char является беззнаковым. Если же нужно хранить в одном байте целое число со знаком, то используется тип signed char.

Обычно в языках программирования есть основной целочисленный тип int, занимающий 4 байта. Тип short (короткий) должен быть в два раза меньше, а тип long (длинный) – в два раза больше. Но сложилось так, что в языке С тип long занимает столько же памяти, что и int, и фактически ничем не отличается от него. Для исправления этой ситуации в стандарте С11 появился тип long long.

Вещественные типы данных предназначены для хранения чисел, которые могут быть дробными. Например, длина окружности, вес товара и т. д.

Имеется три вещественных типа: float (занимает 4 байта), double (8 байт) и long double (10 байт) – новый тип в стандарте С99. При записи вещественного числа в память оно нормализуется: в нем выделяется мантисса и порядок, которые хранятся отдельно (см. рисунок 1.12).

Например, 8320.4 представляется в виде $0{,}83204 \cdot 10^4$. При этом 0,83204 – мантисса, а 4 – порядок.

В типе float для хранения мантиссы обычно выделяется 23 бита, для порядка – 8 бит. Поскольку старшая цифра мантиссы всегда равна 0, она не хранится.

Для типа double для мантиссы отводится 52 бита и для порядка – 11 бит.

Очевидно, что размер памяти для мантиссы определяет точность числа, а размер памяти для порядка – его диапазон.

Диапазоны возможных значений вещественных типов показаны в таблице 2.2. При этом нужно учитывать, что соответствующий диапазон имеется и для отрицательных значений. Числа, близкие к 0, не различаются компилятором. Например, для типа float числа в диапазоне от $-3{,}4 \cdot 10^{-38}$ до $3{,}4 \cdot 10^{-38}$ считаются равными 0 (рисунок 2.1).

Таблица 2.2 – **Вещественные типы языка С++**

Тип данных	Выделяемое количество байт памяти	Диапазон значений
float	4	3.4e – 38 … 3.4e + 38
double	8	1.7e – 308 … 1.7e + 308
long double	10	3.4e – 4932 … 3.4e + 4932

Символьные типы данных предназначены для хранения одного символа, закодированного с помощью некоторого двоичного числа. Для типа char выделяется один байт памяти, в который можно записать 256 различных кодов.

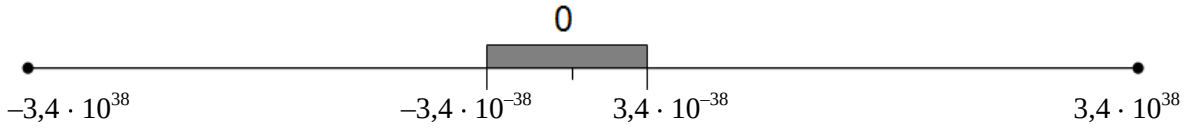


Рисунок 2.1 – **Диапазон значений для типа float (заштрихованная область не различается компилятором)**

При записи символа в память или его воспроизведения на экран система пользуется кодовой таблицей. Пример системы кодировки представлен на рисунке 2.2.

sp 32	! 33	" 34	# 35	$ 36	% 37	& 38	' 39	(40	) 41	* 42	+ 43	, 44	- 45	. 46	/ 47
0 48	1 49	2 50	3 51	4 52	5 53	6 54	7 55	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	^ 94	_ 95
` 96	a 97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	y 121	z 122	{ 123	| 124	} 125	~ 126	

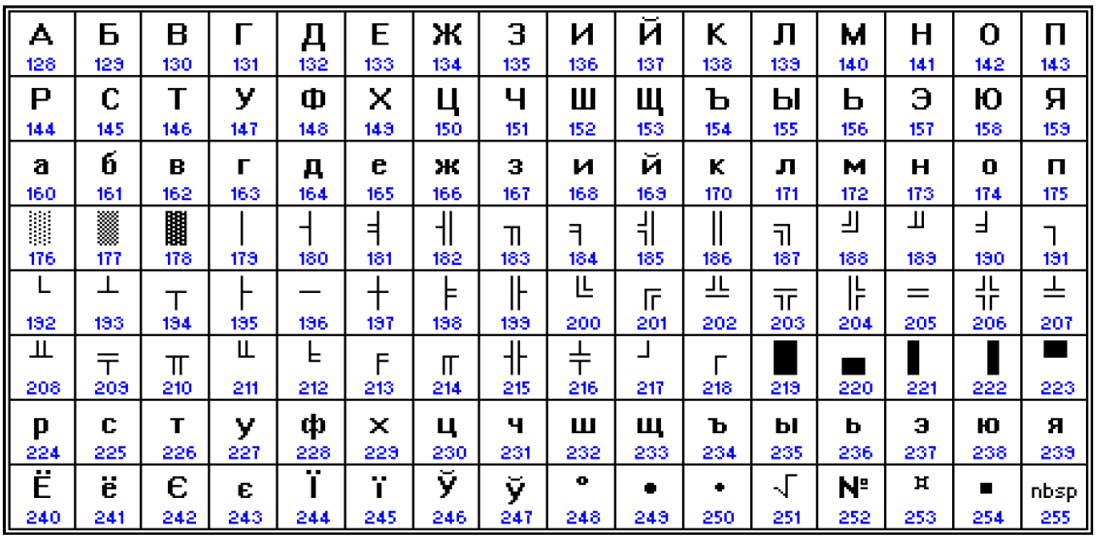


Рисунок 2.2 – **Кодовая таблица MS DOS (866)**

Первые коды от 0 до 31 предназначены для служебных (непечатаемых) символов, поэтому в таблице они не приведены. Нужно понимать, что кроме кодировки, представленной на рисунке 2.2, существуют и другие кодовые таблицы. Обычно первая часть этих таблиц совпадает, а вторая может реализоваться по-разному. Во вторую часть таблицы входят символы национальных алфавитов. В нашем случае используется кириллица, а могут быть, например, арабские буквы.

Когда используем функцию setlocale(), то фактически настраиваем программу на использование другой кодовой таблицы.

Язык С позволяет рассматривать символьный тип также как целое число, размещенное в одном байте.

В 1991 г. была предложена новая система кодирования Unicode, в которой символ занимает 2 (или даже 4) байта. Для использования такой кодировки в стандарт С99 введены типы char16_t и char32_t (или wchar_t). Unicode позволяет закодировать одновременно все национальные алфавиты и поэтому решает проблему «кракозябр» при выводе текста. В данном пособии будем пользоваться традиционными однобайтными символами типа char.

Логический тип данных bool предназначен для хранения данных логического типа. Он может иметь значение true (истина) или false (ложь) Для хранения значения выделяется 1 байт.

Приведенное представление чисел реализовано для микропроцессоров Pentium. Для других платформ внутреннее представление типов данных может отличаться. Операция sizeof() позволяет получить размер типа или переменной в байтах для данной конкретной реализации. Аргументом этой операции может быть как название типа, так и имя переменной:

```
#include <iostream>
using namespace std;
void main()
{
      int i;
      cout <<sizeof(i) <<"\n"; //выводится 4
      cout <<sizeof(long) <<"\n"; //также выводится 4
      cout <<sizeof(double) <<"\n"; //выводится 8
      system("pause");
}
```

2.2. Описание переменных и констант

Любая переменная или константа должна быть описана в программе перед ее использованием. Причем в языке С описание переменной может располагаться в любом месте программы (в отличие, например, от Паскаля).

При описании переменной указывается ее тип и имя. Можно также после знака «=» указать начальное значение переменной (или значение константы). Когда компилятор обрабатывает оператор описания переменной, он выделяет для нее память. Если начальное значение не задано, то часто невозможно предугадать, чем заполнена эта память (фактически она заполняется мусором, который остался на этом месте в памяти от предыдущих каких-то данных).

Для константы дополнительно указывается ключевое слово «const». Константа обязательно должна быть инициализирована, и далее в программе ей нельзя ничего присваивать (это неизменяемое значение).

Например:

- `int k=0; //объявление целой переменной и задание начального значения;`
- `const double Pi=3.1415; //объявление вещественной константы;`
- `char ch='A'; //символьная переменная получает начальное значение;`
- `bool flag; //булевская переменная не получила начального значения;`

…

```
//Pi=8.6;//ошибка!
```

Важное значение имеет место размещения описания переменной в тексте программы, поскольку оно задает *область действия* имени.

Если переменная определена внутри блока (блок ограничен фигурными скобками), она называется *локальной*, область ее действия – от точки описания до конца блока, включая все вложенные блоки. Если переменная определена вне любого блока, она называется *глобальной*, и областью ее действия считается файл, в котором она определена, от точки описания до его конца.

Примечание – Глобальные переменные нужно стремиться использовать как можно реже. Чем уже область действия переменной, тем проще поиск ошибок в программе.

Кроме области действия, имя переменной обладает также областью видимости. *Область видимости* имени – это часть текста программы, в которой имя можно явно использовать. Чаще всего область видимости совпадает с областью действия. Исключением является ситуация, когда во вложенном блоке определена переменная с таким же именем, что и у переменной вне блока. В этом случае внешняя переменная во вложенном блоке невидима, хотя он и входит в ее область действия. Имя переменной должно быть уникальным в своей области видимости.

В примере ниже символьная переменная *simv* глобальная, и она доступна во всем файле. А целая переменная *k* описана два раза: сначала просто в функции main, а потом во вложенном блоке. Объявление *k* внутри блока делает «внешнюю» переменную *k* недоступной. После выхода из блока описание «внутренней» переменной *k* становится недействительным (эта переменная уничтожается при выходе из блока).

```
#include <iostream>
using namespace std;
char simv='Y'; //глобальная переменная
void main()
{
     setlocale(LC_ALL,"rus");
     int k=0; //локальная переменная
     {
          cout<<"k= "<<k<<"\n"; //вывод k=0
          int k=3; //объявление этой переменной делает невидимой
                   // переменную k, описанную ранее
          cout<<"k= "<<k<<"\n"; //вывод k=3
          cout<<"simv= "<<simv<<"\n"; //вывод 'Y’
     }
     cout<<"k= "<<k<<"\n"; //вывод k=0, снова доступна
     system("pause");
}
```

2.3. Литералы

Литералы – это константы, которые не имеют имени. Тем не менее они имеют тип, поскольку компилятор должен размещать их в памяти и знать, сколько именно байт памяти выделить. Тип литерала определяется по его внешнему виду.

Так, обычное целое число в программе имеет по умолчанию тип int. Если по каким-то причинам этот тип не устраивает программиста, то можно указать тип с помощью суффиксов L, l (long) и U, u (unsigned). Например, константа 32L будет иметь тип long и занимать 4 байта. Можно использовать суффиксы L и U одновременно, например, 22UL или 5Lu.

Целый литерал может быть также записан в восьмеричном или шестнадцатеричном виде. Для этого перед числом указываются префиксы:

- 0 – восьмеричное число;
- 0x (0X) – шестнадцатеричное число.

Двоичные литералы (с префиксом 0b или 0B) появились только в стандарте C++14 (вышел в августе 2014 г.).

Вещественные литералы имеют по умолчанию тип double. При этом число может быть как в экспоненциальном формате, так и в формате с плавающей точкой. Можно явно указать тип литерала с помощью суффиксов F, f (float) и L, l (long). Например, литерал 2E+6L имеет тип long double, а 3.14f – тип float.

Символьный литерал представляет собой один символ в одинарных кавычках (апострофах). Например, 'R' или '*'.

Булевский литерал бывает только двух видов: true (истина) и false (ложь).

Кроме того, в языке C имеется строковый литерал (хотя строкового типа данных нет). Это последовательность символов, заключенных в двойные кавычки, например, "Visual".

Примеры задания литералов:

```
#include <iostream>
using namespace std;
void main()
{
	setlocale(LC_ALL,"rus");
	long i=28l; //28l – литерал типа long
	i=5; //5 -литерал типа int
	cout<<"Восьмеричное число "<<057<<"\n"; //литерал типа int в
                                                //восьмеричном формате
	cout<<"Шестнадцатеричное число "<<0x1A<<"\n";//литерал типа int в
                                                        //шестнадцатеричном формате
	double x;
	x=3.14*i; //3.14 – литерал типа double
	x=3.14f*i; //3.14f – литерал типа float
	cout<<"Вещественное число: "<<2.148E2<<"\n";//литерал типа double
	cout<<"Это и есть строковый литерал:\n";
	bool flag=true; //true – литерал булевского типа
	char ch='%'; //'%' – литерал символьного типа
	system("pause");
}
```

На рисунке 2.3 показано окно консоли, которое будет получено при запуске этой программы. Следует заметить, что потоковый вывод по умолчанию все равно выводит в десятичной системе счисления и в формате с фиксированной точкой.

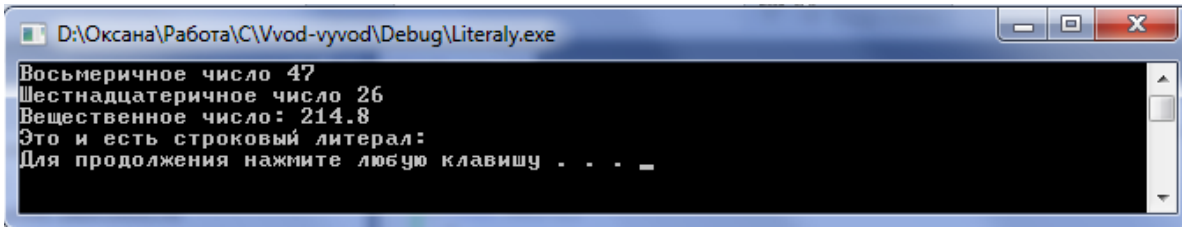


Рисунок 2.3 – **Окно консоли при выводе литералов**

2.4. Потоковый ввод и вывод

Потоковый ввод и вывод – это элемент языка C++, а не классического языка C. Для использования потокового ввода и вывода необходимо подключить заголовочный файл библиотеки <iostream> и открыть соответствующее пространство имен:

```
#include <iostream>
using namespace std;
```

Для *вывода* используется оператор вставки в поток <<. Поток, связанный с экраном консоли, называется cout. Оператор вставки можно использовать несколько раз в одной строке:

```
cout<<"Значение суммы: "<<Sum<<"\n";
```

В этом примере сначала выводится на консоль строковый литерал, затем значение переменной *Sum*, а потом снова строковый литерал, содержащий один символ, – перевод курсора на новую строку. В поток можно выводить также не только значение переменной, но и результат вычисления выражения.

Для *ввода* используется оператор вставки в противоположном направлении >>, а поток ввода, связанный с консолью (т. е. с клавиатурой), называется cin. Здесь также можно вводить несколько значений последовательно: `cin>>a>>b;`.

Вводимые таким образом значения должны разделяться одним или несколькими пробелами, а ввод завершается после нажатия клавиши «Enter». Дробная часть вещественных чисел отделяется десятичной точкой, как и в литералах программы.

При вводе значения в переменную старое ее значение «затирается», перестает существовать.

При вводе чисел принято сначала выводить приглашение для пользователя, например:

```
#include <iostream>
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     double a,b;
     cout<<"Введите два числа: ";
     cin>>a>>b;
     cout<<"Сумма чисел равна: "<<a+b<<"\n";
     system("pause");
}
```

Результат работы этой программы показан на рисунке 2.4.

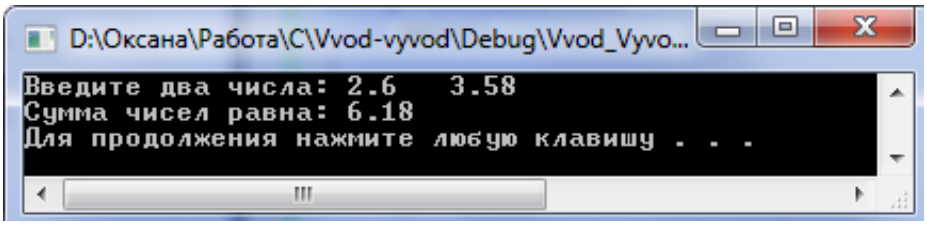


Рисунок 2.4 – **Окно консоли для программы сложения двух чисел**

2.5. Выражения и операции

Выражения состоят из операндов, знаков операций и скобок. Каждый операнд является, в свою очередь, выражением или одним из его частных случаев, например, константой или переменной.

Примеры выражений:

- `(a + 0.12)/6;`
- `x&&y||!z;`
- `a = b = c.`

Операции задают действия, которые необходимо выполнить. В зависимости от количества операндов выделяют три вида операций:

- *Унарные* (один операнд). Например: *a*++; –*b*. Причем знак операции может стоять и слева, и справа от операнда.
- *Бинарные* (два операнда). Например: *a* + *b*, *x*/*y*.
- *Тернарная* (три операнда). Например: *a* > *b*?*a* : *b*.

Список возможных операций языка С приведен в таблице 2.3 (в порядке убывания приоритета). Группы операций с одинаковым приоритетом отделяются двойными линиями.

Таблица 2.3 – **Операции языка С**

Операция	Краткое описание
Унарные операции	
++	Инкремент
– –	Декремент
!	Логическое отрицание
sizeof	Размер объекта или типа
–	Арифметическое отрицание (унарный минус)
+	Унарный плюс
(<тип>)	Преобразование типа
Бинарные и тернарная операции	
*	Умножение
/	Деление
%	Остаток от деления
+	Сложение
–	Вычитание

Окончание таблицы 2.3

Операция	Краткое описание
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
==	Равно
!=	Неравно
&&	Логическое И
||	Логическое ИЛИ
? :	Условная операция (*тернарная*)
=	Присваивание
*=	Умножение с присваиванием
/=	Деление с присваиванием
%=	Остаток от деления с присваиванием
+=	Сложение с присваиванием
–=	Вычитание с присваиванием

Приоритет задает порядок вычисления операций в выражении. То есть, например, в выражении *a* + *b* * 3 сначала выполняется умножение, так как его приоритет выше.

Если в одном выражении записано несколько операций одинакового приоритета, то унарные операции, условная операция и операции присваивания выполняются *справа налево*, остальные – *слева направо*. Например, *a* = *b* = *c* означает *a* = (*b* = *c*), *a* + *b* + *c* означает (*a* + *b*) + *c*. Для изменения порядка выполнения операций используются круглые скобки.

Рассмотрим некоторые операции более подробно.

Операция присваивания является ключевой операцией при работе с переменной. Выглядит она просто как знак «=». При выполнении операции присваивания вычисляется выражение справа от знака «равно» и результат помещается в переменную, записанную слева от него. При этом старое значение переменной теряется, «затирается» новой информацией.

Например, оператор *a* = *a* + 1 означает, что берется старое значение переменной *a*, увеличивается на 1 и результат записывается опять в переменную *a*. (То же самое можно написать проще: a++, но это будет уже не операция присваивания, а операция инкремента.)

Рассмотрим стандартный *алгоритм перестановки значений двух переменных*.

Пусть есть две переменные *a* и *b*, и нужно поменять местами их значения. Какая последовательность операторов выполнит эти действия? Очевидно, что нельзя присвоить *a* значение *b*, так как при этом значение *a* исчезнет, и его нельзя будет использовать. Для перестановки значений переменных используется дополнительная вспомогательная переменная. Назовем ее, например, *tmp*. Последовательность действий будет следующая:

```
tmp=a; // во вспомогательную ячейку запишем копию значения a
a=b; // теперь можем «портить» ячейку a, записав туда значение b
b=temp; //в b записываем копию a, которую ранее сохранили в temp.
```

Наглядно алгоритм представлен на рисунке 2.5.

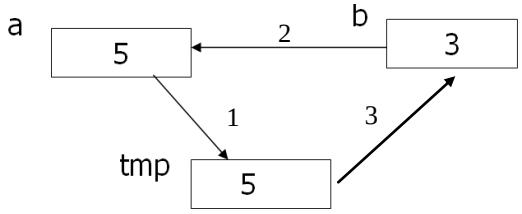


Рисунок 2.5 – **Алгоритм перестановки значений двух переменных**

Арифметические операции:

- –*x* (унарный минус – меняет знак числа на противоположный);
- +*x* (унарный плюс – ничего не делает);
- *x* + *y* (сложение);
- *x* – *y* (вычитание);
- *x***y* (умножение);
- *x*/*y* (деление);
- *x*%*y* (остаток от деления – только для целых операндов).

Выполнение операции деления зависит от типа операндов. Если они имеют вещественный тип, то результат тоже будет вещественным, т. е. будет содержать дробную часть. Если же операнды целые, то результат также будет иметь целый тип (дробная часть результата просто отбрасывается).

Таким образом, результат операции 1 / 2 равен 0 (литералы 1 и 2 по умолчанию целые, поэтому и результат целый – дробная часть будет отброшена). А результат операции 1.0 / 2.0 равен 0.5 (литералы 1.0 и 2.0 по умолчанию имеют тип double, поэтому и результат – вещественное число).

Если операнды имеют разный тип, то перед выполнением операции выполняется преобразование типа. При этом используется общий принцип – преобразование короткого типа к более длинному. (О преобразовании типов подробнее будет изложено далее.)

Операция взятия остатка от деления (деление по модулю) в языке С разрешена только для целых операндов (если операнды не целые, выводится сообщение об ошибке на этапе компиляции).

Для деления по модулю (%) запрещено деление на 0, так как выдается ошибка времени выполнения.

Инкремент и декремент.

Инкремент – это увеличение на 1, а декремент – уменьшение на 1, например:

- *a*++; эквивалентно *a* = *a* + 1;
- *b*– –; эквивалентно *b* = *b* – 1.

Эти операции имеют две формы: префиксную и постфиксную.

В *префиксной* форме знак операции ставится перед именем переменной. В этом случае сначала выполняется увеличение (уменьшение), а затем используется значение переменной:

```
int a=3;
cout<<++a; //выводится 4; значение a равно 4
```

В *постфиксной* форме знак операции применяется после имени. В постфиксной форме сначала используется значение переменной, а потом выполняется увеличение (уменьшение):

```
int a=3;
cout<<a++; //выводится 3; значение a равно 4
```

Сокращенные формы операции присваивания позволяют коротко записать выполнение какой-либо арифметической операции над значением переменной с последующим сохранением результата опять в эту же переменную. Например:

```
• a+=10;        // эквивалентно a=a+10;
• a-=10;        // эквивалентно a=a-10;
• a*=10;        // эквивалентно a=a*10;
• a/=10;        // эквивалентно a=a/10;
• a%=10;        // эквивалентно a=a%10;.
```

Условная (тернарная) операция имеет три операнда и следующий формат:

```
операнд_1 ? операнд_2 : операнд_3
```

Сначала вычисляется значение операнда 1. Результат должен быть такого типа, который можно преобразовать к типу bool. Если он равен true, то результатом выполнения всей условной операции будет значение операнда 2, иначе – операнда 3. Например:

```
• c=(a>b)?a:b; // максимум из двух чисел записывается в переменную c;
• i=(i<n)?i+1:1; //если i<n, i принимает значение на единицу
                 //больше, а если нет, то i становится равно 1.
```

2.6. Математические функции

Описание математических функций находятся в библиотеках <stdlib.h>, <math.h> и <cmath>. В Visual Studio эти библиотеки подключаются по умолчанию. В других средах разработки необходимо подключать эти библиотеки директивой препроцессора #include <…>.

Чтобы узнать, какие еще библиотеки автоматически подключаются, необходимо дважды щелкнуть в *Обозревателе решений* по папке *Внешние зависимости*. Чтобы увидеть содержимое какого-либо файла библиотеки, нужно два раза щелкнуть по имени файла.

По умолчанию значения функций и их аргументы имеют тип double. Вещественные литералы также по умолчанию – double. Поэтому, если нет особых причин экономить память, всегда следует использовать для вещественных чисел тип double.

Список некоторых математических функций приведен в таблице 2.4.

Таблица 2.4 – **Математические функции библиотеки <math.h>**

Функция	Описание
abs(a)	Модуль или абсолютное значение от *a*
sqrt(a)	Корень квадратный из *a*, причем *a* не отрицательное
pow(a, b)	Возведение *a* в степень *b*. Пример: pow (2,3) = 8.0

ceil(a)	Округление *a* до наименьшего целого, но не меньше чем *a* (округление до следующего целого). Примеры: ceil (2.3) = 3.0, ceil (–2.3) = –2.0
floor(a)	Округление *a* до наибольшего целого, но не больше чем *a* (округление до предыдущего целого). Примеры: floor (12.4) = 12.0, floor (–2.9) = –3.0
fmod(a, b)	Вычисление остатка от *a*/*b* для вещественных чисел. Примеры: fmod (4.4, 7.5) = 4.4, fmod (7.5, 4.4) = 3.1
exp(a)	Вычисление экспоненты e^a
sin(a)	Синус (*a* задается в радианах)
cos(a)	Косинус (*a* задается в радианах)
log(a)	Натуральный логарифм *a* (основанием является экспонента)
log10(a)	Десятичный логарифм *a*
asin(a)	Арксинус *a*, где –1.0 < *a* < 1.0

Пример 2.1. Написать программу вычисления значения функции $y = \frac{\sqrt{2}}{2}\sin\frac{x}{2}$.

```
#include <iostream>
using namespace std;
void main()
{    double x,y;
     setlocale(LC_ALL,"rus");
     cout<<"Введите аргумент: ";
     cin>>x;
     y=sqrt(2.)/2*sin(x/2);
     cout<<"Результат функции: "<<y<<"\n";
     system("pause");
}
```

2.7. Преобразование типов

Если операнды некоторой операции имеют различный тип, то перед выполнением этой операции они приводятся к одному типу. Такое преобразование типа может быть неявным (автоматическим) и явным (заданным программистом).

Неявное преобразование типов в выражении (справа от знака «=») всегда является *расширяющим*: более короткие типы приводятся к более длинным. Типы bool, char и short перед выполнением арифметического выражения всегда приводятся к типу int. Ниже показана иерархия типов, согласно которой компилятор выбирает тип, к которому приводятся операнды (и соответственно тип результата):

(bool,char,short)->int->unsigned int->long->unsigned long->float->double->long double.

Например:

```
int a=8;
long b=56;
long c=a+b; //int +long приводитсяк long+long
float x=7.8;
double y=x+3.14; //литерал с точкой по умолчанию double
                 //float+double приводится к double+double.
```

В операции присваивания может неявно выполняться *сужающее* преобразование: если в левой части оператора присваивания более короткий тип, то правая часть преобразуется к этому типу, возможно с потерей данных. При этом компилятор не сообщает об ошибке.

Например:

```
• int a=3.14*2; //сначала в правой части выполняется расширяющее
              //преобразование: double*int->double
              //но потом это число присваивается целой переменной и
 //выполняется сужающее преобразование: отбрасывается дробная часть
```

```
//в результате a получает значение 6;
```
- ```
  int q=-5;
  unsigned int h;
  h=q; //число int записывается в переменную unsigned как набор
  //двоичных цифр. При этом не только теряется знак, но и само число
  //будет записано неправильно: h равно 4294967291.
  ```

Поэтому программисту нужно быть очень внимательным при выполнении сужающих преобразований: вся ответственность за потерю и искажение данных лежит на нем.

Явное преобразование типов задается операцией преобразования одним из двух способов:

- (тип) выражение;
- тип (выражение).

В результате выражение приводится к тому типу, который указал программист.

Например:

- ```
  double z=(double)1/2;    //единица (по умолчанию int)
                           //преобразуется типу double.
  //Аналогично z=1./2
  ```
- ```
  float x=25;
  //int y=x%4;//возникнет ошибка компилятора: нельзя деление по
  //модулю применять к вещественным числам
  int y=(int)x%4; //правильно!
  ```

В сложном выражении преобразование типа выполняется непосредственно перед каждой операцией. Следовательно, последовательность преобразований зависит от порядка выполнения операций.

Пример 2.2.

```
int i=27;
short s=2;
float f=22.3;
bool b=false;
z=i-f+s*b;
```

Определите, какой тип имеет переменная *z*.

Чтобы было проще это обсуждать, запишем вместо переменных их типы:

`int-float+short*bool.`

Сначала будет выполнена операция умножения как более приоритетная. Перед ее выполнением short и bool преобразуются к типу int. Результат будет также int. Получаем: `int-float+int.`

Далее операции одного приоритета выполняются слева направо: в операции разности int приводится к старшему типу float и результат будет float: `float+int.`

Аналогично в операции суммы операнды будут float и результат float. Таким образом, переменную *z* нужно объявить как float (или более длинным старшим типом): `float z=i-f+s*b;.`

Пример 2.3.

```
int a;
float b;
float c=3.5;
b=a=c;
```

Определите, чему станут равны значения переменных *a* и *b*.

Порядок выполнения операции присваивания – справа налево. Поэтому первой будет выполнена операция *a* = *c*. При этом произойдет сужающее преобразование, и дробная часть значения *c* будет отброшена, *a* получит значение 3.

Затем результат этой операции (3) будет присвоен переменной *b*, т. е. она тоже станет равна 3.
```

В случае такой записи оператора присваивания, как *a = b = c*, сначала выполнится *b = c* без потери данных (так как они обе имеют тип float), а уже затем выполняется сужающее преобразование перед присвоением 3,5 целой переменной. В результате *b* равно 3.5, *a* равно 3.

2.8. Списковая инициализация

В стандарте C++11 был добавлен механизм унифицированной (списковой) инициализации, который позволяет задать значение различным программным конструкциям (переменным, массивам, объектам) единообразным способом. Для простых переменных этот способ используется, когда инициализирующее значение берется в фигурные скобки. Знак равенства при этом может быть опущен, например:

- `int n={10}; //в переменную n записывается 10;`
- `char ch{65}; //в переменную ch записывается код символа ‘A’.`

Этот способ инициализации отличается не только синтаксисом записи. Он позволяет защитить код от сужающего преобразования типа и выявить несоответствие типов переменной и инициализирующего значения на этапе компиляции.

Пример 2.4. Рассмотрим такой пример:

```
int x=2.75;
cout<<x;
```

На этапе компиляции никаких ошибок выявлено не будет. А в процессе выполнения программы на консоль будет выведено число 2. Это происходит потому, что во время инициализации было выполнено сужающее преобразование, и дробная часть числа 2.75 была отброшена.

Напишем теперь код с использованием списковой инициализации:

```
int x={2.75};
cout<<x;
```

В этом случае во время компиляции программы будет сгенерирована ошибка, поскольку списковая инициализация защищает от сужающего преобразования типов.

Приведем еще несколько примеров кода, для которых компилятор генерирует ошибку:

- `char sim={300}; //число 300 не входит в диапазон возможных`
  `// значений кодов символов;`
- `short i{70000}; //максимальное значение для этого типа 32 767.`

Если нужно выявлять потенциальные проблемы с потерей данных на этапе компиляции, необходимо использовать списковую инициализацию.

Однако в Visual Studio 2010 Express такой способ инициализации простых переменных не поддерживается. Точнее, можно написать int *x* = {2.75}. Но проверки сужающего преобразования выполнено не будет.

2.9. Ввод и вывод в языке C

В классическом стандарте ANSI C для ввода и вывода использовались функции библиотеки <stdio.h> (в Visual Studio она подключается автоматически). Основное преимущество этих функций – возможность форматирования данных (задать тип представления данных, количество десятичных цифр после запятой, общее количество символов при выводе числа, выравнивание по левому или правому краю и т. д.).

Для *вывода* используется функция printf следующего вида:

```
printf("управляющая строка",список аргументов);
```

Управляющая строка обязательно должна присутствовать, а вот список аргументов может отсутствовать. В списке аргументов может быть любое количество элементов.

Управляющая строка содержит объекты трех типов: обычные символы, которые просто выводятся на экран консоли, escape-последовательности (управляющие символы) и спецификации преобразования.

Каждая *спецификация преобразования* начинается со знака «%» и заканчивается некоторым символом формата (таблица 2.5). Между ними могут встречаться (или не встречаться) знаки, уточняющие формат.

Таблица 2.5 – **Спецификации преобразования функции printf**

Спецификация преобразования	Описание
%d	Десятичное целое число
%o	Восьмеричное целое число
%x	Шестнадцатеричное целое число
%u	Десятичное целое число без знака
%f	Вещественное десятичное число в формате с плавающей точкой (например, 82.312)
%e	Вещественное десятичное число в экспоненциальном формате (например, 0.82312E2)
%g	Вещественное десятичное число (выбирается наиболее короткий формат %f или %e)
%c	Символ
%s	Строка символов
%p	Указатель

Значения аргументов, указанных далее в функции в списке аргументов, при выводе подставляются на место соответствующих спецификаций преобразования в порядке их следования. Английское название спецификации преобразования – «place holder», т. е. «держатель места», поскольку они как бы «бронируют» место для размещения очередного выводимого элемента в строке.

На рисунке 2.6 показан пример вывода на экран консоли значений трех переменных с соответствующим поясняющим текстом. На место первой спецификации %4d подставляется значение первой переменной в списке (*i*), причем выводится она как целое число и размещается в четырех позициях; на место второй спецификации %c ставится значение второй переменной с (символа), а на место спецификации %f – значение вещественной переменной *g*.

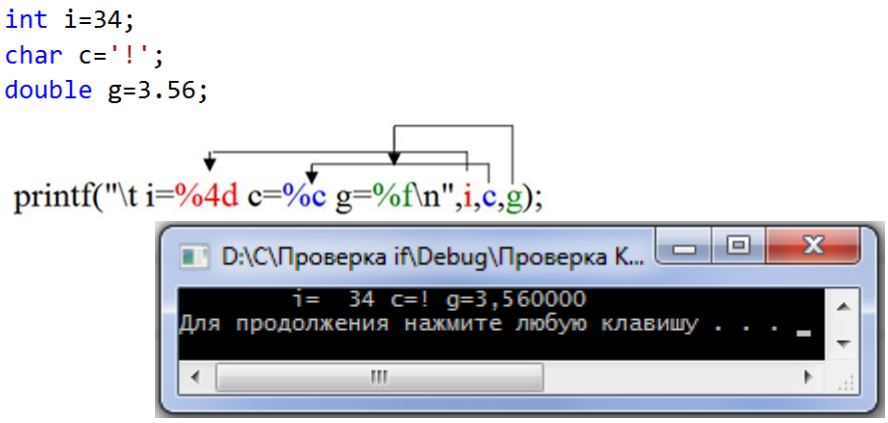


Рисунок 2.6 – **Пример использования управляющей строки в функции printf**

Между знаком % и символом формата могут находиться следующие элементы.

Знак «–» указывает на то, что для выводимого параметра должно быть выравнивание влево в своем поле (по умолчанию выравнивание вправо).

Число, задающее минимальный размер поля. Если аргумент занимает меньше позиций, чем указывает это число, то свободные позиции заполняются пробелами (справа или слева в за-

висимости от выравнивания). Если же аргумент фактически должен занимать больше позиций, чем указано этим числом, то под него отводится столько, сколько нужно (рисунки 2.7 и 2.8).

```
int k=30;
printf("|%4d|\n|%-4d|\n", k, k);
```

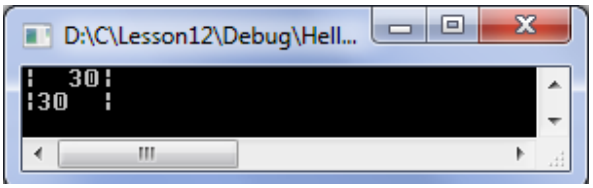


Рисунок 2.7 – **Задание ширины поля и выравнивание**

```
int l=55, j=22556;
printf("|%4d|\n|%4d|\n",l,j);
```

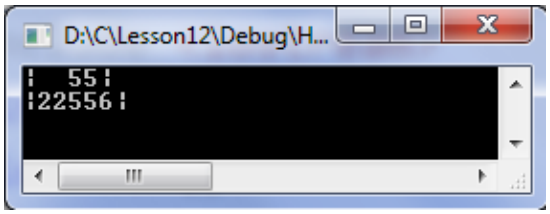


Рисунок 2.8 – **Задание минимальной ширины поля**

Точка отделяет размер поля от количества цифр, которое нужно вывести после десятичной точки (для вещественных типов).

Число после точки – количество знаков после десятичной точки. Если фактическое число знаков больше, то число округляется, если меньше – дополняется нулями (рисунок 2.9).

Если указывается минимальный размер поля, то он рассчитывается для всего числа (включая знак и десятичную точку). Например, для рассмотренного примера можно было бы получить то же самое, указав:

```
printf("x=%10.6f\n",x);
double x=-56.489;
printf("x=%9.2f\n",x);
printf("x=%.6f\n",x);
```

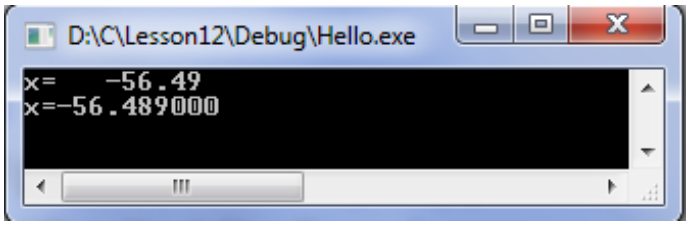


Рисунок 2.9 – **Задание числа знаков после десятичной точки для формата %f**

Поскольку символ «%» имеет служебный смысл, для вывода просто этого символа его нужно продублировать: %% (рисунок 2.10).

```
printf ("Результат 100%%\n");
```

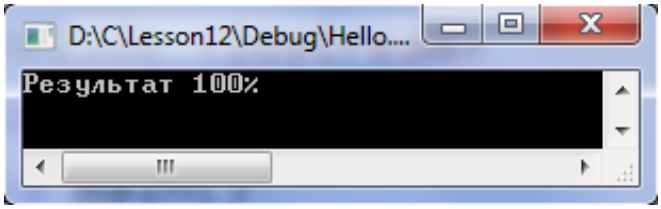


Рисунок 2.10 – **Вывод символа процента**

Этот рисунок также иллюстрирует, что списка аргументов может и не быть.

При использовании современного компилятора, поддерживающего средства работы с двухбайтовыми символами (в кодировке Unicode), можно к спецификации %c применить модификатор l, чтобы указать на использование двухбайтовых символов. Модификатор l можно также использовать со спецификацией %s для вывода строки двухбайтовых символов.

Кроме того, модификатор l можно поставить перед командами форматирования вещественных чисел e, f, и g. В этом случае он уведомит о выводе значения типа long double.

В версии C99 добавлен еще ряд спецификаций форматирования. В заключение отметим, что функция printf возвращает в качестве результата количество символов, выведенное на консоль. (В примерах выше не использовался этот факт.)

Для *форматного ввода* используют функцию scanf, которая формально описывается следующим образом:

```
scanf("управляющая строка",список указателей на переменные);
```

Функция scanf должна считать текст из консоли, преобразовать его в данные нужного типа и разместить их в соответствующие ячейки памяти. Поэтому аргументы функции scanf должны быть указателями на соответствующие переменные. Пока можно представлять, что указатель – это адрес переменной в памяти. Операция получения указателя на переменную (взятие адреса переменной) обозначается знаком «&».

Управляющая строка содержит спецификации преобразования (таблица 2.6) и используется для установления количества и типа аргументов, которые ожидаются при вводе.

Таблица 2.6 – **Спецификаторы преобразования функции scanf**

Спецификация преобразования	Описание
%d	На входе ожидается десятичное целое число
%o	На входе ожидается восьмеричное целое число
%x	На входе ожидается шестнадцатеричное целое число
%u	На входе ожидается десятичное целое число без знака
%lld	На входе ожидается целое число типа long long

Окончание таблицы 2.6

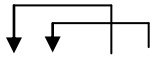
Спецификация преобразования	Описание
%f	На входе ожидается появление вещественного числа типа float
%lf	На входе ожидается появление вещественного числа типа double
%p	На входе ожидается появление указателя (шестнадцатеричного числа)
%c	На входе ожидается появление одиночного символа
%s	На входе ожидается появление строки символов

В управляющую строку могут также включаться символы табуляции и перехода на новую строку (все они игнорируются), а также обычные символы, кроме «%» (считается, что они должны совпадать с очередными символами во входном потоке). На начальном этапе не рекомендуется включать в управляющую строку ничего, кроме спецификаций преобразования.

Как и в случае вывода, устанавливается соответствие между спецификацией преобразования и аргументом в списке указателей в порядке их следования (рисунок 2.11).

Обратите внимание, что при использовании функции scanf нужно при вводе дробную часть числа отделять запятой, а не точкой.

```
float y;
int k;
printf("Введите значения y и k через пробел: ");
```



```
scanf("%f%d",&y,&k);
```

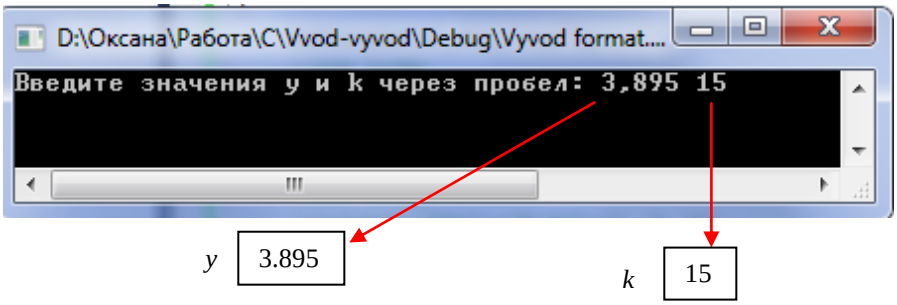


Рисунок 2.11 – **Использование форматного ввода**

Элементы в потоке ввода разделяются одним или несколькими пробелами. Ввод завершается нажатием клавиши «Enter». После выполнения функции scanf() курсор автоматически переходит на новую строку.

Между знаком «%» и символом форматирования можно указать знак «*» (запрещение присваивания). Это означает, что очередное значение будет считано из потока ввода, но в память не запишется, т. е. пропускается (рисунок 2.12).

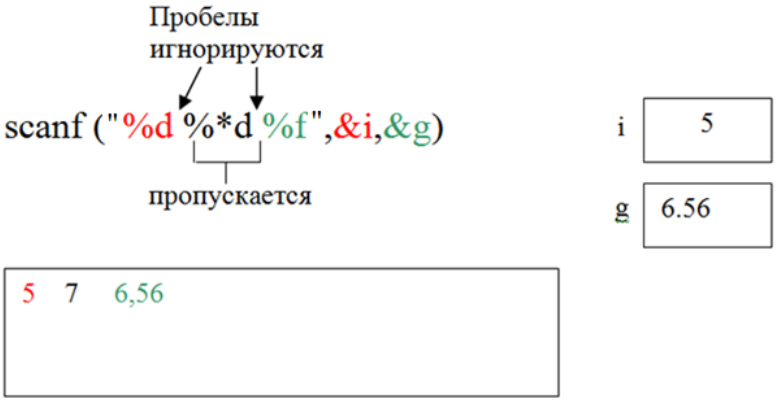


Рисунок 2.12 – **Пропуск элемента в потоке ввода**

Ввод символа и строки в функции scanf() связан с определенными проблемами. Он будет подробно рассматриваться, когда речь пойдет о строках. Пока рекомендуется использовать эту функцию только для ввода чисел.

В Visual Studio 2013 и более поздних версиях (в зависимости от настроек) компилятор может отказаться работать с функцией scanf и потребовать использовать вместо нее более безопасную функцию scanf_s. Эта функция имеет дополнительный параметр – размер буфера (необязательный), позволяющий проконтролировать ввод и избежать изменения области памяти, которая вводимому объекту не принадлежит. Например, при вводе строки эта функция контролирует, чтобы вводимые данные не вышли за пределы памяти, отведенной под эту строку. Для ввода чисел размер буфера можно не указывать, а использовать эту функцию так же, как и scanf.

Важным побочным следствием функции scanf_s является то, что здесь дробная часть вещественного числа при вводе отделяется точкой, а не запятой.

Задачи

Задача 2.1. Пользователь вводит с клавиатуры две стороны прямоугольника. Определить площадь и периметр прямоугольника.

Задача 2.2. Пользователь вводит с клавиатуры три числа. Посчитать их сумму, произведение и среднее арифметическое.

Задача 2.3. Пользователь вводит с клавиатуры время в секундах. Необходимо написать программу, которая определит, сколько введенный интервал времени содержит часов, минут и секунд. Например, 5 311 с = 1 ч 28 мин 31 с.

Задача 2.4. Зарплата менеджера составляет 100 долл. США плюс 5% от продаж. Пользователь вводит с клавиатуры общую сумму сделок менеджера за месяц. Посчитать итоговую зарплату менеджера.

Задача 2.5. Вычислить значения функций и вывести их на экран (результаты расчетов по этим функциям должны совпадать):

- $Z_1 = 2\sin^2(3\pi - 2a)\cos^2(5\pi + 2a)$;
- $Z_2 = \frac{1}{4} - \frac{1}{4}\sin\left(\frac{5}{2}\pi - 8a\right)$.

Для самостоятельного решения

Задача 2.6. Пользователь вводит с клавиатуры символ. Вывести на экран его код. Использовать форматированный вывод.

Задача 2.7. Пользователь вводит с клавиатуры объем флешки в гигабайтах. Посчитать, сколько на нее поместится фильмов размером 760 Мб, музыкальных клипов объемом 95 Мб, музыкальных композиций объемом 7 Мб или текстовых документов объемом 350 Кб. Пересчитать результат, учитывая, что сначала пользователь записывает фильмы, пока для них есть место. После того, как фильмы уже не помещаются (но осталось еще место), он записывает клипы. После того, как ни один клип также уже не помещается на флешку, на оставшееся место записывается музыка, а свободное место заполняется текстовыми документами.

Задача 2.8. Пользователь вводит с клавиатуры денежную сумму в рублях и копейках (рубли и копейки вводятся в разные переменные). Сумма может быть введена правильно (например, 19 р. 90 к.) и неправильно (например, 22 р. 978 к.). Написать программу, которая, используя только линейный алгоритм, осуществит корректировку введенной денежной суммы в правильную форму. Например, если пользователь ввел 11 р. 150 к., программа должна вывести на экран сумму 12 р. 50 к. Использовать форматированный вывод.

Задача 2.9. Написать программу, которая преобразует введенное с клавиатуры дробное число в денежный формат. Например, число 12,518 должно быть преобразовано к виду 12 р. 52 к. Использовать функцию printf().

Задача 2.10. Пользователь вводит с клавиатуры положительное дробное число. Округлить его до двух знаков после запятой и вывести на экран с помощью потокового вывода. Необходимо предусмотреть округление по правилам, т. е. число 12,341 округлялось до 12,34, а число 12,349 – до 12,35.

ТЕМА 3. ОПЕРАТОРЫ ВЕТВЛЕНИЯ

3.1. Операции сравнения

Операции сравнения, или отношения, позволяют сравнить две величины (таблица 3.1).

Таблица 3.1 – **Операции сравнения (отношения)**

Операция	Утверждение
<	Левый операнд меньше, чем правый
>	Левый операнд больше, чем правый
<=	Левый операнд меньше или равен правому

>=	Левый операнд больше или равен правому
==	Левый операнд равен правому
!=	Левый операнд не равен правому

Результат этих операций имеет булевский тип (true – истина, false – ложь), например:

- 3 < 6 – результат true;
- 3 > 6 – результат false;
- 3 <= 3 – результат true;
- 3 >= 6 – результат false;
- 3 == 6 – результат false;
- 3! = 6 – результат true.

Обратите внимание на операцию проверки на равенство. Два знака «==» используются потому, что один такой знак указывает на операцию присваивания. Начинающие программисты часто путают эти операции.

Например, если написать if (a = 4) {…}, то вместо *сравнения* значения *a* и числа 4 переменной *a* будет *присвоено* значение 4 (с потерей предыдущего значения), а результат этого действия будет true, поскольку произойдет преобразование результата операции (4) в булевский тип. В результате действия в фигурных скобках будут выполнены всегда, независимо от того, какое значение имела переменная *a* до этого.

Правила преобразования числовых типов в тип bool (и обратно) в языке С следующие:

1. Если выполняется неявное преобразование к типу bool, то любое значение, не равное 0 (даже отрицательное), трактуется как true, а 0 считается false.

2. При обратных преобразованиях величин типа bool к целому типу значение true преобразуется в целую константу 1, а значение false – в 0.

Например:

- `cout<<(3>6);//выводится 0;`
- `cout<<(3<6);//выводится 1.`

3.2. Логические операции

Логические операции выполняются над аргументами булевского типа. К ним относятся операции && («логическое И»), || («логическое ИЛИ») и ! («логическое НЕ»). Принято описывать действие этих операций с помощью таблиц истинности.

Операция «логическое И» дает результат *true*, если оба операнда *true*, т. е. и то, и другое утверждение истинны (таблица 3.2).

Таблица 3.2 – **Логическое И**

a	*b*	*a*&&*b*
true	true	true
true	false	false
false	true	false
false	false	false

Операция «логическое ИЛИ» дает результат *true*, если хотя бы один из операндов *true*, т. е. истинно одно или другое утверждение либо оба сразу (таблица 3.3).

Таблица 3.3 – **Логическое ИЛИ**

a	*b*	*a*||*b*
true	true	true
true	false	true
false	true	true
false	false	false

Операция «логическое НЕ» дает результат, противоположный значению аргумента: меняет true на false и наоборот (таблица 3.4).

Таблица 3.4 – **Логическое НЕ**

a	!*a*
true	false
false	true

В сложном логическом выражении эти операции выполняются слева направо в порядке их приоритетов. Напомним, что наивысший приоритет имеет унарная операция ! (НЕ), далее операция && (И – логическое умножение), а затем || (ИЛИ – логическое сложение). Операции сравнения имеют приоритет ниже чем !, но выше чем && и ||. Поэтому следующие выражения эквивалентны:

($x < y$) && ($k == 2$) || ($k > 10$) и $x < y$ && $k == 2$ || $k > 10$.

То есть скобки в этом случае не обязательны. Но лучше их поставить во избежание ошибок.

Если при выполнении логической операции значения первого операнда достаточно, чтобы определить результат операции, второй операнд не вычисляется.

Пример 3.1. Рассмотрим логические выражения:

- $4 < 10$ || $k > 3$;
- ($i < n$) && ($a > 0$).

В первом случае поскольку первый операнд операции || true, то результат будет true независимо от значения второго операнда. Поэтому он даже не вычисляется.

Во втором случае сначала проверяется условие ($i < n$), и если оно ложно, то второй аргумент операции && не вычисляется.

Приведем примеры вычисления логических выражений:

```
bool a=true, b=false, c=true, d=false;
a&&b||c&&d      //результат false;
a&&(c||!b);     //результат true;
(4<10)&&c||b;   //результат true.
```

3.3. Условный оператор if

При описании синтаксических конструкций языка обычно пользуются следующими неформальными правилами:

- Необязательные части синтаксических конструкций заключаются в квадратные скобки.
- Текст, который необходимо заменить конкретным значением, пишется по-русски.
- Выбор одного из нескольких элементов обозначается вертикальной чертой.
- Фигурные скобки используются для группировки элементов, из которых требуется выбрать только один.

Например, пусть описание функции выглядит следующим образом:

[{ void | int }] имя().

Это означает, что сначала может идти одно из ключевых слов: void или int (вертикальная черта и фигурные скобки). Но может и вообще ничего не быть (квадратные скобки). Обязательны лишь имя функции (произвольный идентификатор языка С) и круглые скобки сразу после имени.

Пользуясь этим способом, опишем синтаксис условного оператора

```
if (выражение) оператор_1; [else оператор_2;]
```

После ключевого слова if (если) следует выражение в круглых скобках. Тип этого выражения должен приводиться к булевскому типу. Если значение выражения true (истина), то выполняется оператор 1. Если значение false (ложь), то выполняется оператор 2 после слова else.

Причем ветки else может и не быть (об этом говорят квадратные скобки). Считается, что в этом случае оператор if имеет сокращенный вид.

Блок-схемы полной и сокращенной формы условного оператора показаны на рисунке 3.1.

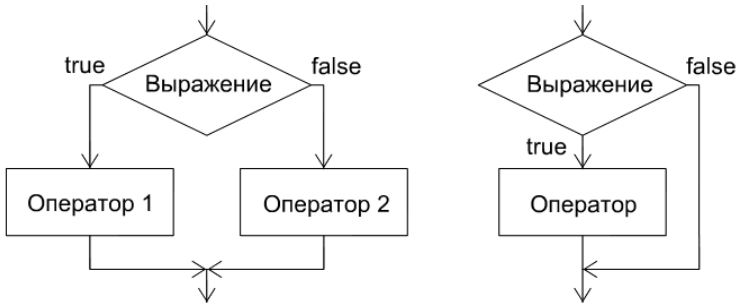


Рисунок 3.1 – **Блок-схемы условного оператора if**

После выполнения любой из веток оператора if управление передается на следующий после него оператор.

Если нужно, чтобы по какой-то ветке (когда выражение истинно или когда ложно) выполнялся не один оператор, а несколько, то их объединяют в составной оператор с помощью фигурных скобок.

Простой пример использования условного оператора – корректное выполнение операции деления. После ввода делимого и делителя нужно проверить, равен ли делитель 0. Если это так, то выводится сообщение о невозможности деления. Если же нет, вычисляется выражение и результат выводится на консоль:

```
#include <iostream>
using namespace std;
void main()
{
	setlocale(LC_ALL,"rus");
	double x,y,z;
	cout<<"Введите делимое:";
	cin>>x;
	cout<<"Введите делитель:";
	cin>>y;
	if(y==0)
		cout<<"На 0 делить нельзя!\n";
	else
	{
		z=x/y;
		cout<<"x/y="<<z<<"\n";
	}
	system("pause");
}
```

В этом примере в случае, если условие истинно, выполняется один оператор вывода на консоль. А по ветке else должно выполняться два оператора: присваивания и вывода. Поэтому они берутся в фигурные скобки.

Обратите внимание на правильную структурную запись программы: те операторы, которые выполняются по каждой ветке условного оператора, сдвинуты вправо на 3–4 позиции (или на знак табуляции). Фигурные скобки, ограничивающие составной оператор (блок), расположены строго друг под другом. Таким образом, отражается тот факт, что одни операторы «вложены» в другие.

Если же операторы должны выполняться последовательно, то сдвигать их друг относительно друга не нужно.

Блок-схема алгоритма этой программы показана на рисунке 3.2.

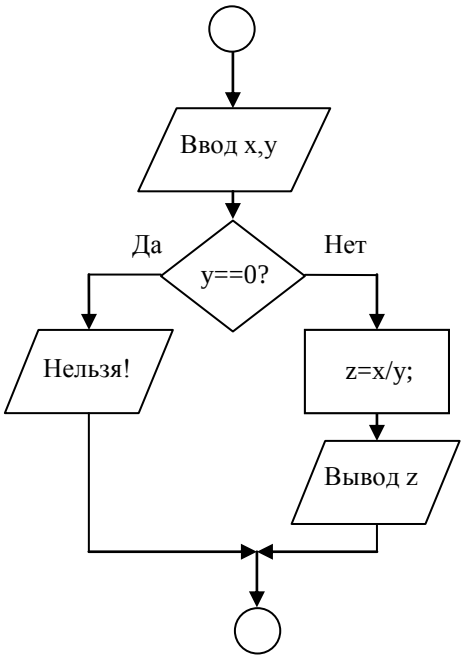


Рисунок 3.2 – **Блок-схема расчета частного от деления**

Эту же задачу можно решить с помощью сокращенной формы условного оператора. В этом случае проверяется условие отличия от 0 делителя. Если оно истинно, то деление выполняется и результат выводится на экран. Если же ложно, действие закончено. Например:

```
#include <iostream>
using namespace std;
void main()
{
	setlocale(LC_ALL,"rus");
	double x,y,z;
	cout<<"Введите делимое:";
	cin>>x;
	cout<<"Введите делитель:";
	cin>>y;
	if(y!=0)
	{
		z=x/y;
		cout<<"x/y="<<z<<"\n";
	}
	system("pause");
}
```

Этот вариант решения, очевидно, хуже предыдущего. Требуемое действие (деление) не выполняется, а пользователь не получает никакой информации о причине этого. Блок-схема данного алгоритма отражена на рисунке 3.3.

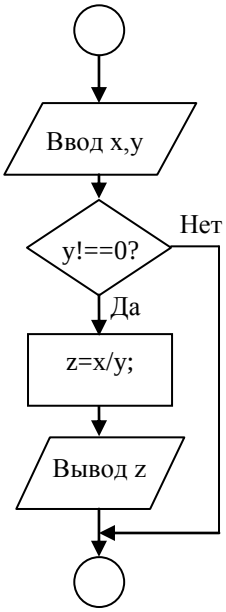


Рисунок 3.3 – **Использование сокращенной формы оператора if**

Распространенной ошибкой новичков является попытка вставить несколько операторов перед else, не объединив их с помощью фигурных скобок:

```
if(y!=0)
     z=x/y;
     cout<<"x/y="<<z<<"\n";//ОШИБКА!Разрыв конструкции if-else
else
     cout<<"На 0 делить нельзя!\n";
```

Чтобы не возникало таких ошибок, на начальном этапе обучения программированию лучше всегда брать ветки if и else в фигурные скобки независимо от того, сколько операторов должно быть записано по этой ветке (один или больше). То есть сначала создается конструкция

```
if(условие)
{
}
else
{
}
```

Затем записываются операторы внутри фигурных скобок.

Выражение в скобках может иметь не только тип bool, но и любой целый тип. В этом случае выполняется неявное приведение результата выражения к типу bool по правилам, описанным в разделе 3.1: любое целое число, не равное 0, считается true. Если же выражение имеет нулевое значение, то результат false.

Таким образом, записать условие, что сумма чисел не равна нулю, можно так:

```
if (a+b) {…}
```

А можно и так:

```
if (a+b!=0) {…}
```

Второй способ предпочтительнее, т. е. является более понятным и очевидным.

Рассмотрим еще примеры преобразования к типу bool в условном операторе:

```
int b=0;
if(++b)
     cout<<"Test successful\n";
```

В этом примере используется префиксный инкремент: сначала увеличивается значение переменной *b*, а потом используется. То есть переменная *b* становится равной 1, что преобразуется к типу bool как true. Поэтому сообщение «Test successful» на консоли появится.

```
int b=0;
if(b++)
     cout<<"Test successful\n";
```

А в этом случае инкремент постфиксный: сначала значение переменной *b* используется для проверки условия, а потом уже увеличивается на 1. То есть в момент проверки условия значение *b*, равное 0, преобразуется к false, и вывода на консоль сообщения «Test successful» не состоится.

В качестве оператора, выполняемого в случае истинности или ложности условия, заданного в скобках, может стоять любой оператор языка C, в том числе и оператор if. Таким образом, операторы if могут быть вложены друг в друга.

Пример 3.2. Пользователь вводит сумму, которую он должен заплатить за товар. Приложение рассчитывает сумму, которую он должен внести в кассу с учетом скидки. Скидка рассчитывается по следующим правилам:

- если покупатель приобретает товар на сумму более 100 тыс. р., то он получает скидку 5%;
- если же сумма товара больше, чем 500 тыс. р., то размер скидки – 10%;
- если же сумма покупки более 1 млн р., то скидка будет 25%.

Разумеется, скидка рассчитывается один раз: если сумма больше 1 млн р., то она больше и 500 тыс. р., но второй раз скидка уже не должна вычисляться. Поэтому начнем проверку с самой большой границы суммы. А следующее условие скидки проверяется только в том случае, если предыдущая проверка оказалась неудачной:

```
#include <iostream>
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     double sum, pay;
     cout<<"Введите сумму покупки в тысячах рублей: ";
     cin>>sum;
     if(sum>1000)
          pay=sum-sum*0.25;
     else if(sum>500)
               pay=sum-sum*0.1;
          else if(sum>100)
                    pay=sum-sum*0.05;
               else
                     pay=sum;
     cout<<"Вам нужно заплатить: "<<pay<<" тысяч рублей\n";
     system("pause");
}
```

Получили «лесенку» вложенных if-else. Обратите внимание, что else всегда относится к предыдущему if и записывается строго под ним. Блок-схема основной части программы показана на рисунке 3.4.

Да Нет

Да Нет

Да Нет

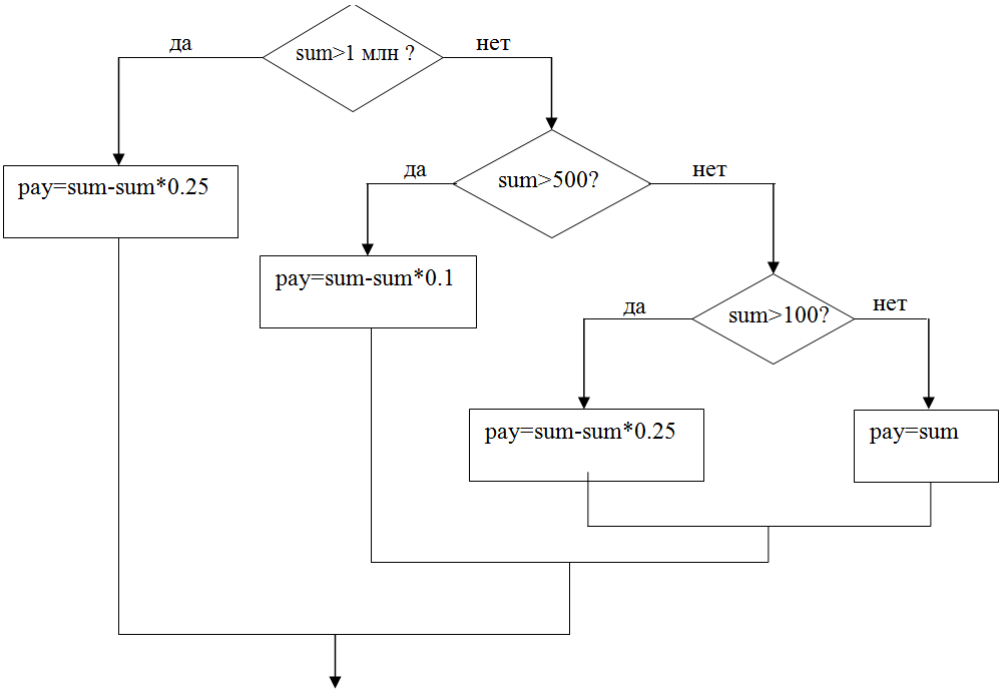


Рисунок 3.4 – **Блок-схема алгоритма вычисления скидки**

Следует отметить, что использование вложенных операторов if очень сильно усложняет программу, а программист всегда должен стремиться к простоте и очевидности своего кода.

Попробуем модифицировать этот пример, не используя вложенные if. Для этого начнем проверку с самой маленькой суммы. Количество денег, которое нужно заплатить (переменная *pay*), будет перезаписываться (с «затиранием» предыдущего результата) каждый раз, когда обнаруживаем, что сумма оказалась подходящей под следующий уровень скидки:

```
#include <iostream>
using namespace std;
void main()
{
	setlocale(LC_ALL,"rus");
	double sum, pay;
	cout<<"Введите сумму покупки в тысячах рублей: ";
	cin>>sum;
	pay=sum; //нет скидки
	if(sum>100)
		pay=sum-sum*0.05; //скидка 5%
	if(sum>500)
		pay=sum-sum*0.1; //отменяем скидку 5%, назначаем 10%
	if(sum>1000)
		pay=sum-sum*0.25;//отменяем скидку 10%, назначаем 25%
	cout<<"Вам нужно заплатить: "<<pay<<" тысяч рублей\n";
	system("pause");
}
```

Блок-схема этого варианта алгоритма отражена на рисунке 3.5.

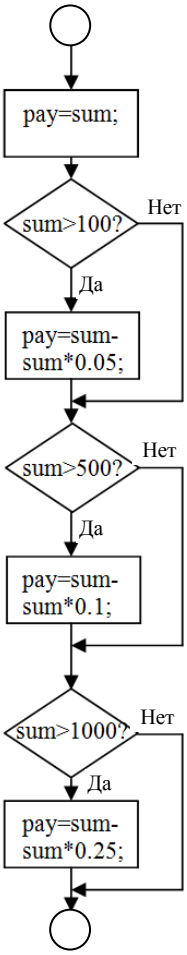


Рисунок 3.5 – **Блок-схема вычисления скидки без вложенных if**

При сравнении вещественных чисел на равенство следует знать, что из-за погрешности представления вещественных значений в памяти эта операция может быть выполнена некорректно. Нужно ее избегать. Для этого лучше сравнивать модуль разности с некоторым малым числом:

```
float a, b;
…
if( a == b ) {… }                    // не рекомендуется!
if( abs(a - b) < 1e-6 ) {… }         // надежно!
```

Значение величины, с которой сравнивается модуль разности, следует выбирать в зависимости от решаемой задачи и точности участвующих в выражении переменных.

Для иллюстрации этой проблемы рассмотрим пример программы, в которой демонстрируется, что $\frac{1}{3}+\frac{4}{3}\neq\frac{5}{3}$:

```
#include <iostream>
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     double three=3.0;
     double x,y,z;
     x=1/three;
     y=4/three;
     z=5/three;
     if(x+y==z) printf("1/3 + 4/3 == 5/3\n");
     else printf("1/3 + 4/3 != 5/3\n");
     system("pause");
}
```

Результат работы программы показан на рисунке 3.6.

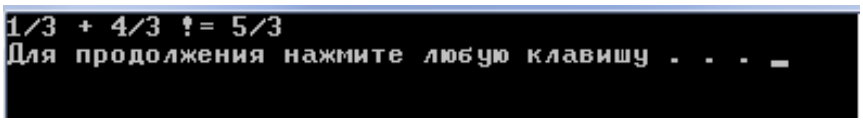


Рисунок 3.6 – **Результат работы программы, использующей сравнение вещественных чисел на равенство**

3.4. Оператор выбора

Оператор выбора (переключатель) используется в тех случаях, когда необходимо разветвить процесс выполнения программы на несколько направлений. Формат оператора следующий:

```
switch(выражение)
{
     case значение 1: действия 1; break;
     case значение 2: действия 2; break;
      …
     case значение n: действия n; break;
     [default: действия по умолчанию;]
}
```

Выполнение оператора начинается с вычисления выражения в скобках. Оно должно иметь целочисленный тип (или тип, приводимый к целочисленному, например, char). Затем последовательно проверяются все значения, указанные после слова «case». Если находим совпадение со значением вычисленного выражения, то управление передается операторам, указанным после двоеточия на соответствующей ветке. Все константы после слова «case» должны иметь разные значения.

Если же не найдено ни одного совпадения, то выполняются операторы после слова «default». Ветка default может отсутствовать, тогда в случае отсутствия совпадений с константами case не выполняются никакие действия. Блок-схема оператора выбора показана на рисунке 3.7.

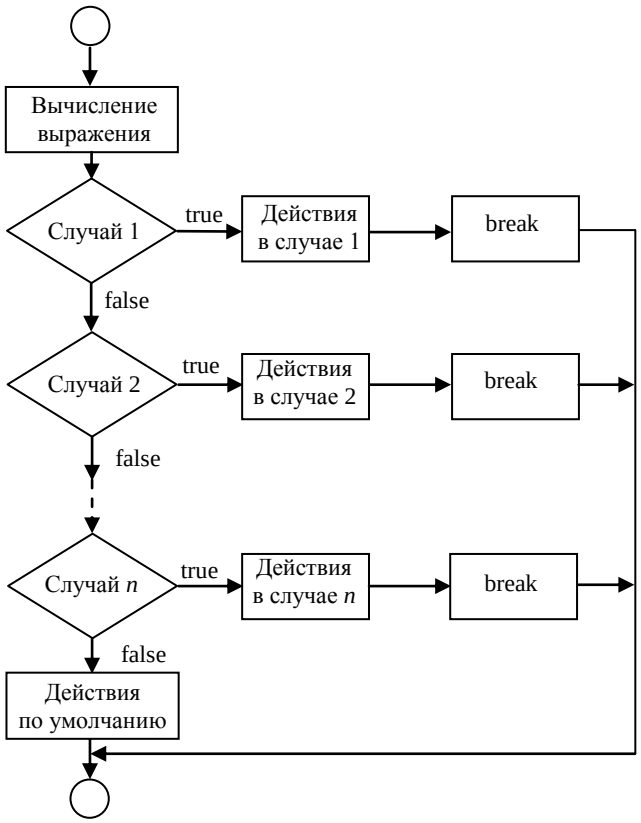


Рисунок 3.7 – **Блок-схема оператора выбора switch**

В конце каждой ветки следует указывать оператор break (выход). Он переводит управление на следующий после switch оператор, т. е. выводит за пределы конструкции выбора. Если оператор break не указан, то после выполнения действий выбранной ветки будут выполняться и все последующие действия до конца оператора switch (включая и действия по умолчанию).

Нужно отметить, что оператор break не является частью конструкции выбора. Он может использоваться и в других контекстах (например, для экстренного выхода из цикла).

После операторов последней группы (после default или после последнего case) оператор break можно не указывать.

Ветка default в принципе может располагаться в любом месте конструкции switch. Однако хорошим стилем программирования считается поместить ее в конце этого оператора.

Примечание – Обратите внимание на оформление программы: действия по каждой ветке не обязательно заключать в фигурные скобки. Если количество операторов, выполняемых по каждой ветке, мало, то их можно записать в одну строку. Если же по каждой ветке операторов много или они имеют сложную структуру, то лучше их размещать друг под другом, используя сдвиг вправо на 3–4 позиции (для указания вложенности).

Пример 3.3. Требуется реализовать простейший калькулятор, выполняющий четыре арифметических действия.

Знак операции имеет тип char и используется в качестве выражения, по которому производится выбор. Если введенный знак не является знаком сложения, вычитания, умножения или деления, то выводится сообщение «неизвестная операция» (действия по ветке default). В случае, если это знак деления, проверяется второй операнд. При равенстве его 0 выводится сообщение о невозможности выполнения операции:

```
#include <iostream>
using namespace std;
void main()
{
	setlocale(LC_ALL,"rus");
	int a, b, res;
	char op;
	cout <<"\n Введите 1-й операнд : ";
	cin >> a;
	cout <<"\n Введите знак операции : ";
	cin >> op;
	cout <<"\n Введите 2-й операнд : ";
	cin >> b;
	bool flag = true; //признак вывода результата
	switch (op)
	{
		case'+': res = a + b; break;
		case'-': res = a - b; break;
		case'*': res = a * b; break;
		case'/':
			if(b!=0) res = a / b;
			else
			{
				cout<<"Деление на 0 невозможно!\n";
				flag=false; //результат выводить не нужно
			}
			break;
		default :
			cout <<"\n Неизвестная операция\n";
			flag = false; //результат выводить не нужно
	}
	if (flag) cout <<"\n Результат : "<< res<<"\n";
	system("pause");
}
```

В данной программе применен один из известных приемов программирования – использование флага. Флаг – это булевская переменная. В данной программе она означает необходимость вывода результата операции на экран. Перед началом оператора switch флаг устанавли-

вается в true. Если по ходу анализа знака операции выясняется, что результат не может быть рассчитан (неизвестный знак или деление на 0), то флаг устанавливается в false (сбрасывается). Вывод результата на экран выполняется только в случае установленного флага. Такое использование булевской переменной позволяет избежать дублирования кода (например, вывода результата в каждой ветке оператора switch).

Пример 3.4. Программа-конвертор валют.

Пользователь вводит сумму в белорусских рублях, а программа рассчитывает соответствующую сумму в долларах, евро или российских рублях. Причем желаемый вариант перевода запрашивается у пользователя.

В решении используются курсы валют, установленные на 20 ноября 2014 г. (доллар США – 10 885 бел. р., евро – 13 630, российский рубль – 235 бел. р.).

```
#define DOLLAR 10885
#define EURO 13630
#define RUS 235
#include <iostream>
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     float sum,rezult;
     cout<<"Введите сумму в бел. рублях: ";
     cin>>sum;
     cout<<"Введите D, если хотите получить сумму в долларах США\n";
     cout<<"Введите E, если хотите получить сумму в евро\n";
     cout<<"Введите R, если хотите получить сумму в рос. рублях\n";
     cout<<"Ваш выбор: ";
     char v;
     cin>>v;
     switch (v)
     {
          case'D':
          case'd': cout<<"Вам следует получить "<<sum/DOLLAR<<"$\n";
                   break;
          case'E':
          case'e': cout<<"Вам следует получить "<<sum/EURO<<"EUR\n";
                   break;
          case'R':
          case'r': cout<<"Вам следует получить "<<sum/RUS<<"RUR\n";
                   break;
          default: cout<<"Неверный выбор\n";
     }
     system("pause");
}
```

В этой программе нужно обратить внимание на два важных момента:

• Использование двух меток case подряд: пользователь мог ввести букву «d» в любом регистре (прописную или строчную) – программа отработает одинаково. Это происходит потому, что по ветке «case 'D'» не содержится оператора break, т. е. управление передается далее на следующую ветку.

• Использование нового типа константы, задаваемой с помощью команды препроцессора `#define`:

```
#define DOLLAR 10885
```

Константа DOLLAR получает значение 10885. Препроцессор обрабатывает текст программы *до начала компиляции*, заменяя во всем тексте программы слово «DOLLAR» на число 10885. Работа препроцессора – это работа с текстом. Таким образом, не происходит никакого выделения памяти под константу. К тому моменту, когда компилятор начнет свою работу, в тексте программы не будет никакого объекта с именем DOLLAR, а будет только число 10885.

Можно было также задать константу уже известным способом:

```
const int DOLLAR=10885;
```

В этом случае была бы выделена область памяти, в которую занесено соответствующее значение.

Какой способ задания констант в программе лучше, сказать сложно. Можно лишь отметить, что в случае использования `#define` происходит экономия памяти (хотя и незначительная в данном примере).

Хороший стиль программирования – записывать символические константы, определяемые через `#define`, прописными буквами.

3.5. Перечисляемый тип данных

Перечисляемый тип данных (enum) используется для улучшения читабельности программ. По сути, этот тип задает набор констант, каждой из которых соответствует целое число. Формат оператора объявления перечисляемого типа следующий:

```
enum [имя_типа] { Константа_1, Константа_2, …}[имя_переменной];
```

Как видно из этого описания, здесь может отсутствовать имя типа или имя переменной (но не оба сразу).

По умолчанию первой константе задается значение 0, а каждой следующей – на единицу больше. Например:

```
enum days {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY} weekday;
```

Здесь объявлен тип days, который является перечисляемым типом. Так же объявлена переменная этого типа `weekday`, которая может принимать значения символических констант `SUNDAY`, `MONDAY`, и т. д. При этом для компилятора `SUNDAY` – это число 0, `MONDAY` – число 1 и т. д.

Можно символическим константам назначить и другие значения, указав их после знака равенства. Если при этом некоторым константам значения явно не заданы, то для них действует правило по умолчанию (на единицу больше предыдущего). Например:

```
enum days {SUNDAY=7, MONDAY=1, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY} weekday;
```

Константа `SUNDAY` получила значение 7, `MONDAY` – значение 1, а `TUESDAY` имеет значение 2 (по умолчанию) и т. д. Таким образом, получим нумерацию дней недели, принятую у нас в стране.

Обратите внимание, что сейчас впервые появилась возможность объявить новый тип данных, а не использовать встроенные. После такого описания можно объявить и другую переменную нового типа days:

```
days birthday;
```

Если же в программе переменная данного перечисляемого типа будет только одна, то имя типа можно не вводить, а ограничиться именем переменной:

```
enum {MY, HIS, HER, THEY} mestoim;
```

При использовании перечисляемого типа в программе следует учесть следующее:

• Идентификаторы, используемые при описании типа, являются константами (далее в программе им нельзя присвоить другое значение):

```
MONDAY=5; //нельзя!
```

• Все имена констант должны быть уникальны, а значения их могут быть одинаковыми:

```
enum imya {VADIM = 2, VANYA = 2, SONIA, YULA = 0, DENIS = SONIA + 20} student;
```

• Перечисление – это отдельный тип данных. Константы VADIM, VANIA и т. д. имеют тип imya.

• Перечисления неявно преобразуются в обычные целочисленные типы, но не наоборот:

```
enum signal {ON,OFF} a;
int i=1;
a=ON;
i=a; // верно, i становится равно 0
a=i; //неверно! Разные типы
```

Хорошим стилем программирования является использование букв верхнего регистра (прописных) для символических констант перечисляемого типа. Это будет напоминать программисту, что это константы, и их нельзя изменять.

Пример 3.5. Напишем программу, реализующую виртуальную поездку по торговому центру с использованием перечисляемого типа данных.

В процессе работы этой программы в момент сравнения целого числа floor с константами по веткам case происходит неявное преобразование перечисляемого типа к целому:

```
#include <iostream>
using namespace std;
void main()
{
     setlocale(LC_ALL, "rus");
     // перечисляемый тип для этажей
     enum level {PARKING, SUPERMARKET, HARDWARESTORES, BOUTIQUES, SPORTSPA,
CLUBRESTAURANTBAR};
int floor; //выбор этажа пользователем
cout <<"\n\t$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n\n";
cout <<"\t Добро пожаловать в наш торгово-развлекательный центр MALL!!!\n";
cout <<"\t Предлагаем Вам проехаться в лифте и посетить все этажи!\n\n";
cout <<"\t $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n\n";
cout <<"\n Нажмите кнопку с номером этажа (от 0 до 5): ";
cin >> floor;
switch(floor)
{
     case PARKING:
               cout <<"\a Вы спустились в паркинг!!!\n";
               break;
     case SUPERMARKET:
               cout <<"\a Вы на первом этаже!";
               cout <<"\n Здесь вы можете посетить наш супермаркет и купить \
продукты и товары для дома.\n\n";
               break;
     case HARDWARESTORES:
               cout <<"\a Вы на втором этаже!";
               cout <<"\n Здесь расположились магазины бытовой техники, IT и \
мобильных телефонов.\n\n";
               break;
     case BOUTIQUES:
               cout <<"\a Вы на третьем этаже!";
               cout <<"\n Здесь вас ждет незабываемый шопинг! Одежда, обувь, \
магазины косметики.\n\n";
               break;
     case SPORTSPA:
               cout <<"\a Вы на четвертом этаже!";
               cout <<"\n Здесь вы можете посетить бассейн, каток, \
спортзалы, spa-салон!\n\n";
               break;
```

```
        case CLUBRESTAURANTBAR:
                cout <<"\a Вы на пятом этаже!";
                cout <<"\n Тут вы можете посетить ночной клуб, бар и \
ресторан!\n\n";
                break;
        default: cout <<"Ошибка! У нас только 5 этажей!\n\n";
    }
        system ("pause");
}
```

3.6. Настройка консоли на ввод и вывод русских букв

Использование функции setlocale() позволяет изменить кодировку русских букв консоли только на вывод. Часто бывает необходимо, чтобы кодировка вводимых символов также совпадала с кодировкой тех символов, которые набираются в тексте программы. В рамках 8-битной кодировки символов это можно сделать, если настроить консоль на кодировку Windows 1251. Чтобы это сделать, нужно подключить библиотеку Windows.h командой препроцессора `#include <Windows.h>`, а в тексте программы использовать следующие функции:

- `SetConsoleCP(1251); //настройка кодировки для ввода;`
- `SetConsoleOutputCP(1251); //настройка кодировки для вывода.`

Разумеется, функция setlocale() уже в этом случае не используется. Однако нужно учесть, что в Visual Studio эта кодировка работает только со шрифтом Lucida Console. Поэтому шрифт в консоли нужно изменить следующим образом:

- Вызвать контекстное меню консоли (правой кнопкой мыши щелкнуть по иконке в левом верхнем углу).
- В контекстном меню выбрать пункт *Свойства*.
- На вкладке *Шрифт* выбрать *Lucida Console* → *Ок* (рисунок 3.8).

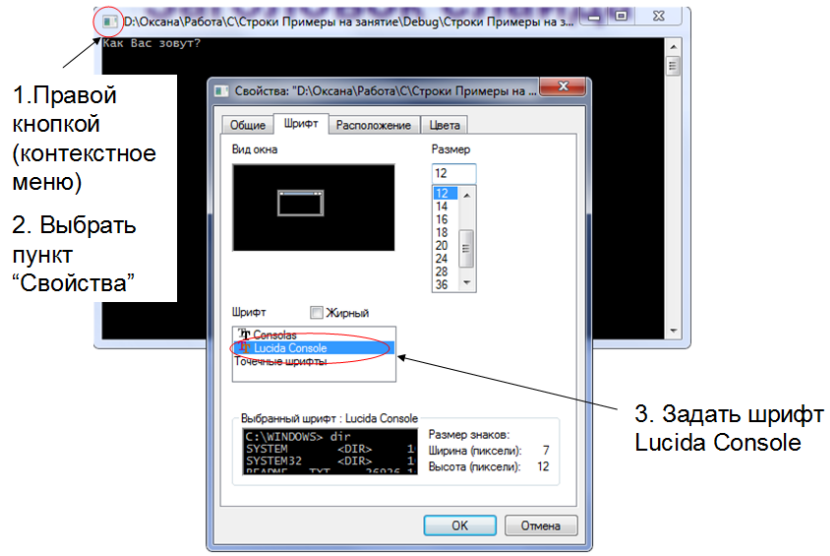


Рисунок 3.8 – **Изменение настроек консоли**

После этого можно быть уверенными, что код символа, набираемого с клавиатуры, код символа, выводимого на экран, и код символа, набираемого в текстовом редакторе Visual Studio, будут одинаковыми.

Если учесть, что в кодировке Windows 1251 русские буквы расположены подряд (сначала прописные, а затем строчные, кроме буквы «ё»), то можно легко проверить, является ли введенный символ русской буквой: достаточно установить, что его код больше буквы «А» и меньше «я». Текст данной программы выглядит так:

```
#include <iostream>
#include <Windows.h>
using namespace std;
void main()
{
     SetConsoleCP(1251);
     SetConsoleOutputCP(1251);
     char c;
     cout<<"Введите символ: ";
     cin>>c;
     if(c>='A'&&c<='я')
          cout<<"Это русская буква\n";
     else
          cout<<"Это не русская буква\n";
     system("pause");
}
```

Задачи

Задача 3.1. Пользователь вводит число. Определить, является ли оно четным.

Задача 3.2. Пользователь с клавиатуры вводит 5 оценок студента. Определить, допущен ли студент к экзамену. Студент получает допуск, если его средний балл 4 и выше.

Задача 3.3. Известно, что 1 дюйм равен 2,54 см. Разработать приложение, переводящее дюймы в сантиметры и наоборот. Диалог с пользователем реализовать через меню.

Задача 3.4. Вычислить значение функции для аргумента, введенного пользователем:

$$y = \begin{cases} 1g^2 2x, \text{ если } x \ge 5; \\ 2x^2, \text{ если } x < -2; \\ \sin x \text{ в остальных случаях.} \end{cases}$$

Задача 3.5. Написать программу, которая принимает от пользователя номер месяца и сообщает время года, к которому этот месяц относится. Задача должна быть решена с использованием конструкции switch.

Для самостоятельного решения

Задача 3.6. Пользователь вводит с клавиатуры 7 чисел. Определить, какое из них является максимальным. (Решение должно быть простым.)

Задача 3.7. Пользователь вводит с клавиатуры целое шестизначное число. Написать программу, которая определяет, является ли введенное число счастливым. (Счастливым считается шестизначное число, у которого сумма первых трех цифр равна сумме вторых трех цифр.) Если пользователь ввел не шестизначное число – вывести сообщение об ошибке.

Задача 3.8. Программа определяет, попадает ли точка с заданными с клавиатуры координатами в область, закрашенную на рисунке 3.9 серым цветом. Величину *R* нужно задать в виде именованной константы. Результат работы программы вывести в виде текстового сообщения.

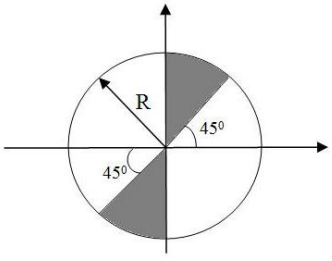


Рисунок 3.9 – **Область на плоскости, для которой выполняется проверка**

Задача 3.9. Грузовой самолет должен пролететь с грузом из пункта A в пункт C через пункт B. Емкость бака для топлива у самолета – 300 л. Потребление топлива самолетом на 1 км в зависимости от веса груза следующее:

- до 500 кг – 1 л/км;
- до 1 000 кг – 4 л/км;
- до 1 500 кг – 7 л/км;
- до 2 000 кг – 9 л/км (более 2 000 кг самолет не поднимает).

Пользователь вводит расстояние между пунктами A и B, расстояние между пунктами B и C, а также вес груза. Программа должна рассчитать, какое минимальное количество топлива необходимо для дозаправки самолету в пункте B, чтобы долететь из пункта A в пункт C. В случае невозможности преодолеть любое из расстояний программа должна вывести сообщение о невозможности полета по введенному маршруту.

Задача 3.10. Написать программу, которая с использованием конструкции switch определяет, каким днем (рабочий день, суббота, воскресенье) является введенный номер дня недели.

Задача 3.11. Пользователь вводит длительность разговора и выбирает направление звонка (Life – Life, Velcom – МТС, МТС – Life и т. п.). Программа должна выводить стоимость такого разговора. Тарифы для всех операторов и для разных направлений можно указать с помощью констант. (Использовать перечисляемый тип.)

Задача 3.12. Пользователь вводит значения трех чисел *x*, *y* и *z*. Рассчитать значение *m* по следующей формуле:

$$m = \frac{max\ (x, y, z)}{min\ (x, y)} + 5.$$

Задача 3.13. Программа принимает от пользователя трехзначное целое число и должна сообщить, состоит ли это число из одинаковых цифр. Если введенное число не трехзначное – сообщить об ошибке.

Задача 3.14. Пользователь вводит целое четырехзначное число. Необходимо поменять в этом числе первую и вторую цифры, а также третью и четвертую цифры. Если число не четырехзначное – сообщить об ошибке.

Задача 3.15. Пользователь вводит с клавиатуры символ. Определить, какой это символ: буква, цифра, знак препинания или другое. (Нельзя пользоваться специальными функциями.)

Задача 3.16. Вася работает программистом и получает 50 долл. США за каждые 100 строк кода. За каждое третье опоздание Васю штрафуют на 20 долл. США.

Реализовать меню:

- Пользователь вводит желаемый доход Васи и количество опозданий. Подсчитать, сколько строк кода ему надо написать.
- Пользователь вводит количество строк кода, написанное Васей, и желаемый объем зарплаты. Подсчитать, сколько раз Вася может опоздать.
- Пользователь вводит количество строк кода и количество опозданий. Определить, сколько денег заплатят Васе (и заплатят ли вообще).

ТЕМА 4. ОПЕРАТОРЫ ЦИКЛА

4.1. Типы циклов

Операторы цикла служат для реализации алгоритмов, которые содержат повторение каких-то однотипных действий. В языке C существует три оператора цикла:

- цикл с предусловием while;
- цикл с постусловием do-while;
- цикл с параметром for.

В теории программирования доказывается, что любой цикл, в принципе, можно организовать только с помощью конструкции while. Однако в практике программирования используются все три типа циклов (в языке C++ их еще больше), поскольку это удобно и делает программу более читабельной.

Пример 4.1. Нужно ввести 10 чисел с клавиатуры и найти их сумму.

Нет необходимости заводить 10 переменных для ввода чисел, а достаточно одной переменной *x*. В эту переменную вводим очередное число, прибавляем его к накапливаемой сумме (переменная s*um*). После этого хранить уже использованное значение *x* нет смысла, поэтому ту же переменную *x* используем для ввода следующего числа.

Таким образом, нам необходимо организовать 10 раз повтор следующих действий:

- Ввод *x*;
- *sum* += *x*.

Конечно, не стоит забывать про инициализацию переменных: в начале программы нужно переменную *sum* очистить, подготовить для накапливания суммы: s*um* = 0.

Чтобы контролировать, сколько повторений сделано, заведем переменную-счетчик *k*. В начале программы еще не было выполнено ни одного повторения, поэтому присвоим этой переменной 0: *k* = 0.

После каждого повторения нужных действий счетчик будем увеличивать на 1, а перед повторением проверять, сколько раз уже выполнены эти действия. Если количество сделанных повторений меньше 10, то нужно войти в цикл и выполнить ввод и увеличение суммы еще раз.

Блок-схема данного алгоритма показана на рисунке 4.1.

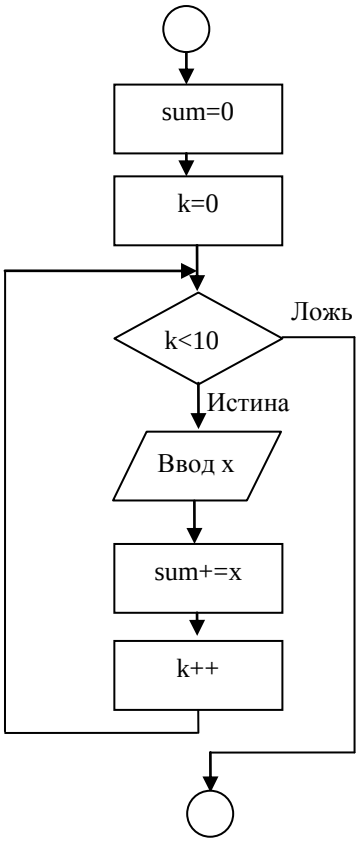


Рисунок 4.1 – **Блок-схема программы суммирования десяти чисел**

4.2. Цикл с предусловием while

Цикл с предусловием while применяется тогда, когда число повторений заранее неизвестно. Проверка условия, которое контролирует число повторений, выполняется до выполнения тела цикла. *Телом цикла* называются те действия, которые нужно повторять многократно. Формат оператора следующий:

```
while (выражение) оператор;
```

При выполнении этого оператора сначала вычисляется выражение и проверяется его истинность. То есть выражение должно иметь булевский тип или тип, который неявно приводится к булевскому. Следует помнить, что любое целое число, не равное 0, автоматически преобразуется в true, а 0 – в false.

Если выражение истинно, то выполняется оператор (тело цикла) и происходит возврат на проверку выражения. Если выражение ложно, то выполнение цикла заканчивается и управление передается на следующий оператор после цикла. Блок-схема цикла while показана на рисунке 4.2.

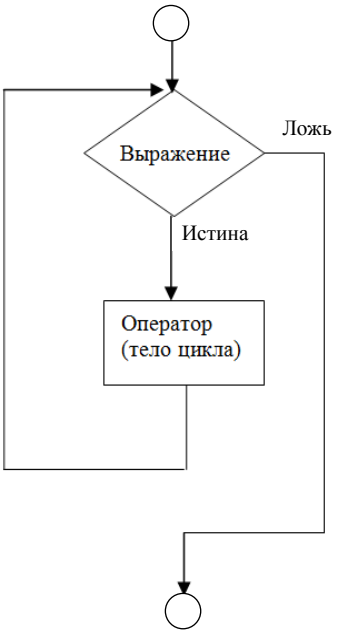


Рисунок 4.2 – **Блок-схема цикла while**

В случае, если условие ложно с самого начала, тело цикла не выполнится ни разу.

Если тело цикло должно состоять более чем из одного оператора, то они объединяются в составной оператор с помощью фигурных скобок.

Правила структурной записи программ требуют сдвигать тело цикла на 3–4 позиции вправо относительно слова «while».

Хотя этот вид цикла больше подходит для случаев, когда число повторений заранее неизвестно, с его помощью можно реализовать цикл в любой ситуации. Например, для вычисления суммы десяти чисел:

```
#include <iostream>
using namespace std;
void main()
{
	setlocale(LC_ALL,"rus");
	int x,sum,k;
	sum=0;
	k=0;
	while(k<10)
	{
		cout<<"Введите число: ";
		cin>>x;
		sum+=x;
		k++;
	}
	cout<<"Сумма чисел= "<<sum<<"\n";
	system("pause");
}
```

4.3. Цикл с параметром for

Этот вид цикла используется в тех случаях, когда заранее известно число повторений. В нем применяется параметр цикла (переменная, которая служит для контроля числа повторений). Формат оператора следующий:

```
for(инициализация; выражение; модификация) оператор;
```

Сначала выполняются действия, указанные в разделе «инициализация». Затем вычисляется значение выражения (условие выполнения цикла) и проверяется на истинность.

Если выражение истинно, то выполняется тело цикла (оператор), затем выполняются действия из раздела «модификация» и происходит возврат на проверку выражения.

Если выражение ложно, то происходит выход из цикла и управление передается на следующий оператор после цикла.

Блок-схема цикла с параметром for показана на рисунке 4.3.

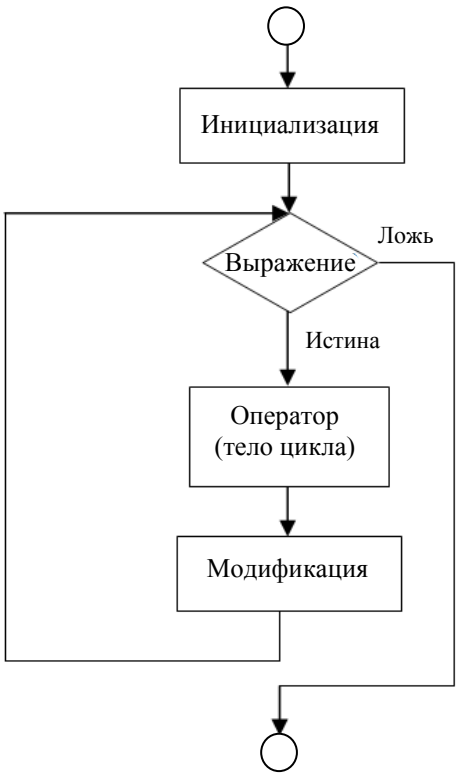


Рисунок 4.3 – **Блок-схема цикла for**

Так же, как и в случае цикла while, тело цикла for может ни разу не выполниться (если выражение ложно с самого начала).

Используем этот вид цикла для записи программы определения суммы десяти чисел:

```
#include <iostream>
using namespace std;
void main()
{
	setlocale(LC_ALL,"rus");
	int x,sum,k;
	sum=0;
	for(k=0;k<10;k++)
	{
		cout<<"Введите число: ";
		cin>>x;
		sum+=x;
```

```
    }
    cout<<"Сумма чисел= "<<sum<<"\n";
    system("pause");
}
```

Обратите внимание, что в теле цикла уже не нужно заботиться об увеличении счетчика: цикл for выполнит это автоматически.

В разделах «инициализация» и «модификация» можно указывать несколько операторов через запятую. Эти операторы выполняются последовательно. Например, можно внести в раздел инициализации обнуление переменной *sum* (хотя это и не совсем правильно с точки зрения хорошего стиля программирования):

```
for(sum=0,k=0;k<10;k++)
{
    cout<<"Введите число: ";
    cin>>x;
    sum+=x;
}
```

Особенности использования цикла for заключаются в следующем:

- Любая из трех компонент цикла for может отсутствовать, но точки с запятыми остаются.
- Если пропущено условие выполнения цикла, то оно по умолчанию считается истинным:

```
for( ; ; ) {…}//бесконечный цикл.
```

- Если переменная объявлена (создана) в разделе инициализации, то после выхода из цикла она перестает существовать:

```
for(int k=0;k<10;k++)
{
    cout<<"Введите число: ";
    cin>>x;
    sum+=x;
}
//после выхода из цикла переменная k не существует.
```

Хотя синтаксис цикла for в языке С достаточно гибкий, рекомендуется придерживаться следующих *правил хорошего стиля программирования*:

- Переменная цикла должна быть целочисленной.
- В разделы модификации и инициализации цикла for помещаются только выражения с управляющими переменными. Манипуляции с другими переменными должны производиться либо перед циклом, либо в теле цикла, например:

```
//плохой код:
int i,sum;
for(i=1,sum=0;i<=10;sum+=i,i++);
//хороший код
int i,sum;
sum=0;
for(i=1;i<=10;i++)
    sum+=i;
```

- Значения управляющей переменной в теле цикла for изменять нельзя:

```
for (int i=1;i<=10;i++)
{
    cout<<i<<“\t”;
    i+=2;              //плохо!
}
```

4.4. Цикл с постусловием do-while

Цикл с постусловием do-while также применяется в том случае, когда заранее не известно число повторений цикла. Формат оператора следующий:

```
do оператор while(выражение);
```

Сначала выполняется оператор (тело цикла), потом вычисляется выражение и проверяется его истинность.

Если выражение истинно, то происходит возврат и тело цикла выполняется еще раз.

Если выражение ложно, то происходит выход из цикла и управление передается следующему за циклом оператору.

Таким образом, тело цикла в этом случае обязательно выполнится хотя бы один раз.

Блок-схема оператора do-while показана на рисунке 4.4.

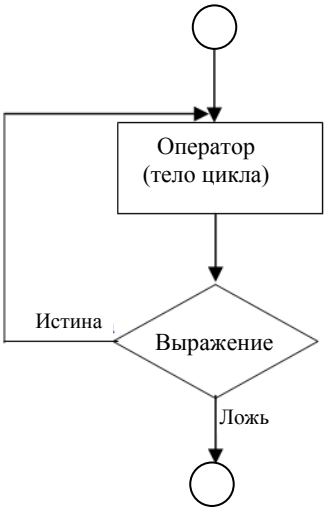


Рисунок 4.4 – **Блок-схема цикла do-while**

Пример 4.2. Запишем пример программы определения суммы десяти чисел с помощью цикла с постусловием do-while:

```
#include <iostream>
using namespace std;
void main()
{
	setlocale(LC_ALL,"rus");
	int x,sum,k;
	sum=0;
	k=0;
	do
	{
		cout<<"Введите число: ";
		cin>>x;
		sum+=x;
		k++;
	}
	while (k<10);
	cout<<"Сумма чисел= "<<sum<<"\n";
	system("pause");
}
```

Конечно, блок-схема программы уже будет несколько другая (рисунок 4.5), хотя результат работы программы абсолютно тот же.

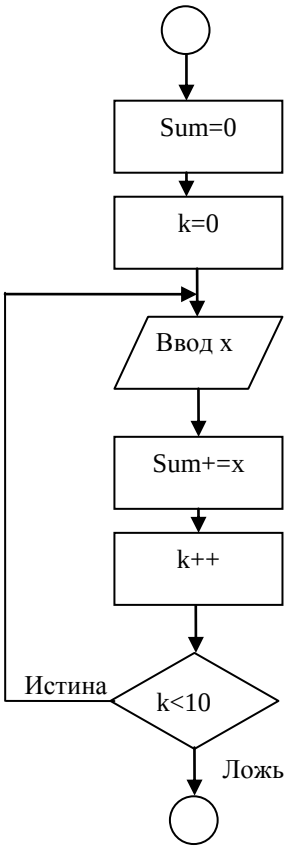


Рисунок 4.5 – **Блок-схема алгоритма суммирования десяти чисел с использованием цикла do-while**

4.5. Использование различных операторов цикла

На примере задачи с определением суммы десяти чисел стало понятно, что один и тот же алгоритм можно записать различными способами. Однако существуют ситуации, когда гораздо удобнее использовать определенный вид цикла. Рассмотрим несколько примеров использования различных операторов цикла.

Пример 4.3. Необходимо найти сумму чисел, вводимых с клавиатуры. Ввод числа 0 означает, что чисел больше нет и нужно вывести результат.

Во-первых, в этой задаче не известно, сколько чисел захочет ввести пользователь. Во-вторых, хотя бы один ввод нужно сделать (как иначе узнать, что чисел нет вообще?). Из этого следует, что лучше всего использовать цикл do-while:

```
#include <iostream>
using namespace std;
void main()
{    setlocale(LC_ALL,"rus");
     double x,sum=0;
     do
     {
          cout<<"Введите число (0 означает окончание процесса)\n";
          cin>>x;
          sum+=x;
     }
     while(abs(x)>1e-10);
     cout<<"Сумма чисел равна= "<<sum<<"\n";
     system("pause");
}
```

Блок-схема программы показана на рисунке 4.6.

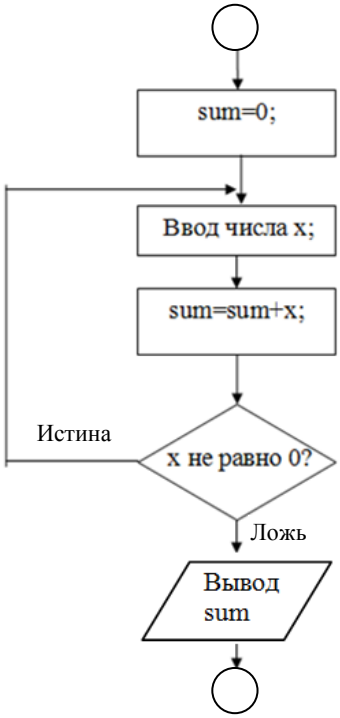


Рисунок 4.6 – **Блок-схема определения суммы неизвестного количества чисел**

Обратите внимание на два момента в этой программе:

• Число, которое вводим, является вещественным. Поэтому лучше не выполнять проверку на неравенство нулю, а сравнить модуль вводимого числа с каким-то очень малым значением (10^{-10}).

• Проверим работу алгоритма в следующей ситуации: пусть пользователь сразу вводит число 0 (т. е. у него нет чисел для ввода). Сумма чисел, которые он не вводил, очевидно, равна нулю. И программа выдаст 0, т. е. сработает правильно. Всегда нужно проверять работу программы на тестах, которые затрагивают «пограничные» ситуации.

Пример 4.4. Вывести таблицу значений функции $y = \frac{\sqrt{2}}{2}\sin\frac{x}{2}$ в каждой точке некоторого интервала. Начальное и конечное значение интервала и шаг табуляции запросить у пользователя.

В начале программы выполним ввод исходных данных. Допустим, левый конец интервала будет при вводе записан в переменную *low*, правый конец – в переменную *high*, а шаг табуляции (приращение аргумента) – в переменную *dx*.

Понятно, что для формирования таблицы аргумент функции (переменная *x*) сначала должен быть равен *low*. Значение *y* рассчитывается по формуле и выводится в строке таблицы на экран (в строке распечатывается значение аргумента *x* и соответствующее значение функции *y*). После этого аргумент увеличивается на шаг табуляции *dx* для повторения расчетов. Это продолжается до тех пор, пока аргумент находится в пределах данного интервала (т. е. меньше или равен *high* – правого конца интервала).

Заранее не известно, сколько получится строк в таблице (сколько раз будет повторяться тело цикла). Если пользователь введет неправильные границы интервала (*high* < *low*), то расчеты выводиться не будут. Поэтому лучше всего для решения этой задачи использовать цикл с предусловием while.

Блок-схема алгоритма примера приведена на рисунке 4.7, а текст программы показан ниже:

```
#include <iostream>
using namespace std;
void main()
{
      double x,y,low,high,dx;
      setlocale(LC_ALL, "rus");
      printf("Введите нижнюю границу переменной x: ");
```

```
        scanf("%lf",&low);
        printf("Введите верхнюю границу переменной x: ");
        scanf("%lf",&high);
        printf("Введите шаг изменения переменной x: ");
        scanf("%lf",&dx) ;
        printf("\n|     x     |     y     |\n") ;
        printf("--------------------------\n");
        x=low;
        while(x<high+1e-9)
        {
             y=sqrt(2.)/2*sin(x/2) ;
             printf("|   %8.3f   |   %8.3f   |\n",x,y) ;
             x+=dx;
        }
        printf("--------------------------\n");
        system("pause");
}
```

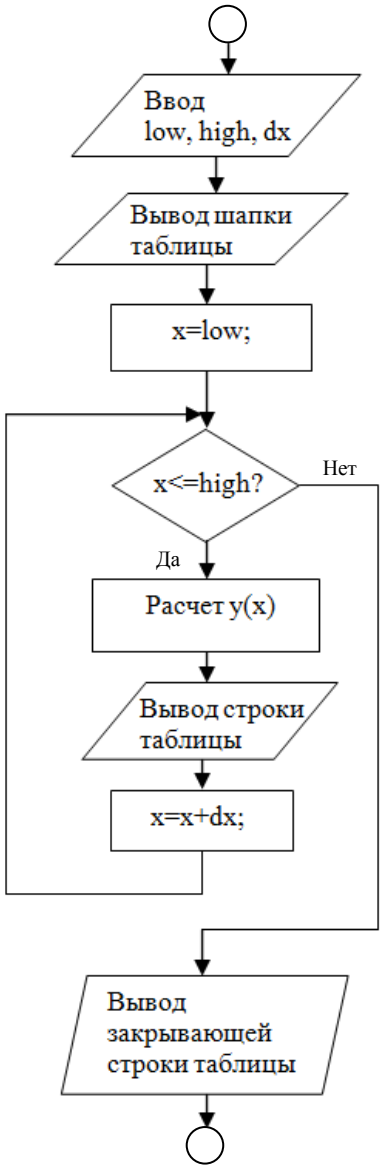


Рисунок 4.7 – **Блок-схема программы табуляции функции**

В этой задаче требуется форматированный вывод, чтобы таблица выглядела аккуратно. Поэтому здесь использовались функции printf и scanf, а не потоковый ввод – вывод.

Казалось бы, условие выполнения цикла должно быть: *x* <= *high*. Но проверка на равенство для вещественных чисел работает плохо, поэтому поставили строгое «меньше», чем *high*

плюс очень маленькое число. Это маленькое число должно быть заведомо меньше любого шага табуляции, который может ввести пользователь. Можно проверить: если написать условие *x* <= *high*, то правая граница интервала не выводится в таблице.

Язык С позволяет решить эту задачу и с использованием цикла for:

```
#include <iostream>
using namespace std;
void main()
{
	double y,low,high,dx;
	setlocale(LC_ALL, "rus");
	printf("Введите нижнюю границу переменной x: ");
	scanf("%lf",&low);
	printf("Введите верхнюю границу переменной x: ");
	scanf("%lf",&high);
	printf("Введите шаг изменения переменной x: ");
	scanf("%lf",&dx) ;
	printf("\n|     x     |     y     |\n");
	printf("-------------------------\n");
	for(double x=low;x<high+1e-9;x+=dx)
	{
		y=sqrt(2.)/2*sin(x/2) ;
		printf("|   %8.3f   |   %8.3f   |\n",x,y) ;
	}
	printf("-------------------------\n");
	system("pause");
}
```

Однако с точки зрения хорошего стиля программирования это решение является не очень грамотным. Считается, что параметр цикла for должен быть целочисленного типа. И хотя синтаксис языка С позволяет реализовать вариант с вещественным параметром цикла, за это его многие критикуют.

4.6. Операторы передачи управления из тела цикла

Для экстренного выхода из цикла применяются два оператора – break и continue.

Оператор break досрочно завершает выполнение цикла (или оператора switch), передавая управление на следующий оператор.

Оператор continue выполняет переход к следующей итерации цикла, пропуская все оставшиеся действия до конца итерации. Если это цикл for, то перед проверкой условия выполняются модификации.

Схема действия указанных операторов показана на рисунке 4.8.

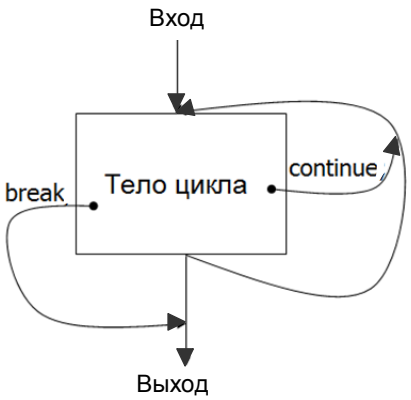


Рисунок 4.8 – **Схема действия операторов break и continue**

Оператор break выполняет выход только из одного цикла (или оператора switch), в котором он непосредственно находится. С его помощью нельзя выйти сразу из всех вложенных циклов.

Хотя операторы break и continue нарушают принципы структурного программирования, иногда они дают изящное решение задачи, плюсы в производительности и улучшают читабельность программ.

Пример 4.5. Пользователь вводит неотрицательные вещественные числа. Ввод любого отрицательного числа означает, что требуется закончить процесс ввода и получить сумму введенных чисел.

Используем бесконечный цикл, задав конструкцию for(;;). Выход из цикла предусмотрим с помощью break в том случае, когда введенное число окажется меньше 0. Оператор *sum += x*, который стоит далее в теле цикла, выполнен уже не будет:

```
#include <iostream>
using namespace std;
void main()
{
     setlocale(LC_ALL, "rus");
     double x,sum=0;
     for(;;)
     {
          cout<<"Введите число (<0 означает окончание ввода )\n";
          cin>>x;
          if(x<0) break;
          sum+=x;
     }
     cout<<"Сумма чисел равна= "<<sum<<"\n";
     system("pause");
}
```

Пример 4.6. Пользователь вводит число. Необходимо распечатать все делители этого числа и найти произведение четных делителей.

Для решения этой задачи можно перебрать все числа, меньшие заданного пользователем (поскольку делитель всегда меньше самого числа). Каждое из этих чисел нужно проверить: если оно является делителем, то распечатать, а если нет, то его просто пропустить. Кроме того, если делитель является четным, то его нужно учесть в произведении (умножить на него переменную *P*, в которой будем накапливать этот результат).

Сначала напишем программу без использования continue:

```
#include <iostream>
using namespace std;
void main()
{    setlocale(LC_ALL, "rus");
     int digit,P=1;
     cout<<"Введите целое число: ";
     cin>>digit;
     cout<<"Делители этого числа: \n";
     for(int i=2;i<digit;i++)
     {
          if(digit%i==0)
          {
               cout<<i<<"\t";
               if(i%2==0)
                    P*=i;
          }
     }
     cout<<"\n Произведение четных делителей= "<<P<<"\n";
     system("pause");
}
```

В теле цикла проверяем, является ли очередное число делителем для digit. Если да, то выводим его на экран и делаем еще проверку на четность.

Аналогичная программа с использованием continue позволяет сразу перейти на следующую итерацию, если обнаружено, что число делителем digit не является:

```
#include <iostream>
using namespace std;
void main()
{    setlocale(LC_ALL, "rus");
     int digit,P=1;
     cout<<"Введите целое число: ";
     cin>>digit;
     cout<<"Делители этого числа: \n";
     for(int i=2;i<digit;i++)
     {
          if(digit%i!=0) continue;
          cout<<i<<"\t";
          if(i%2==0)
               P*=i;
     }
     cout<<"\n Произведение четных делителей= "<<P<<"\n";
     system("pause");
}
```

Этот вариант программы не повышает производительности, но зато упрощает текст программы (уменьшает количество уровней вложенности).

4.7. Операторы goto и return

Оператор goto позволяет передать управление в любую точку программы. Для этого в нужном месте программы перед оператором ставится метка (идентификатор, после которого идет двоеточие), а в goto указывается эта метка:

```
goto dalee;
…
dalee: k++;
```

Метка – это обычный идентификатор, т. е. подчиняется всем правилам создания имен в языке С.

С точки зрения принципов структурного программирования необходимо отказаться от использования этого оператора, поскольку он:

- делает программу сложной для понимания;
- повышает вероятность ошибок;
- не дает возможность компилятору оптимизировать код.

Однако никакие принципы (в том числе принципы структурного программирования) не нужно возводить в абсолют. Существуют ситуации, когда использование оператора goto вполне оправдано. Например, в случае, если нужно выйти из вложенных циклов:

```
for(int i=0;j<m;i++)
     for(int j=0;j<n;j++)
     { …
          if (…) goto myexit;
          …
     }
myexit: cout<<"Продолжение программы…\n"
```

Оператор break тут не поможет, поскольку выведет только из внутреннего цикла.

В любом случае с помощью goto управление не должно передаваться назад по тексту программы. Использовать этот оператор нужно только в крайних случаях, когда другие средства языка программу не упрощают, а усложняют.

Оператор return служит для выхода из функции (передачи управления назад той функции, которая вызвала текущую). Поскольку пока использовалась только одна функция (main), оператор return означает прекращение программы (возврат управления Visual Studio).

Использование оператора return позволяет избежать вложенных конструкций языка и упростить текст программы.

Пример 4.7. Реализуем игру в виде теста. Следующий вопрос должен быть задан пользователю, если предыдущий ответ был правильным. Если же ответ был неверным, то нужно вывести финальное сообщение и закончить программу.

Организуем программу так, чтобы избежать вложенных конструкций if и иметь возможность легко добавлять новые вопросы:

```
#include <iostream>
using namespace std;
void main()
{
	int answer;
	setlocale(LC_ALL,"rus");
	cout<<"Первый вопрос \n";
	cout<<"Варианты ответа: ";
	cin>>answer;
	if(answer!=1)
	{
		cout<<"Финальное сообщение\n";
		system("pause");
		return;
	}
	cout<<"Второй вопрос \n";
	cout<<"Варианты ответа: ";
	cin>>answer;
	if(answer!=2)
	{
		cout<<"Финальное сообщение\n";
		system("pause");
		return;
	}
	cout<<"Победа!\n";
	system("pause");
}
```

4.8. Интегрированный отладчик Visual Studio

Часто ошибки в программе связаны не с синтаксисом, а внутренней логикой программы. Компилятор такие ошибки не обнаруживает, но программа работает не так, как нужно. Такие ошибки обнаруживаются во время тестирования и отладки программы.

Тестирование означает, что программа прогоняется на заранее заготовленных тестах, и результаты ее работы сравниваются с тем, что «должно быть». Тесты должны охватывать как обычные, так и различные «пограничные» ситуации (например, неверный ввод данных пользователем). В принципе, хорошая программа должна быть полностью защищена «от дурака», сбоев оборудования, т. е. работать адекватно в любой ситуации.

Отладка – это поиск ошибки в программе, когда тест показывает, что работает она неверно. Для этой цели и служит интегрированный отладчик Visual Studio. Он дает возможность остановить процесс выполнения программы в любом месте, просмотреть текущие значения переменных, выполнить программу пошагово – оператор за оператором и многое другое. В общем, дает возможность «залезть внутрь» программы и «проследить» за процессом ее выполнения.

Интересно, что английский термин «отладка» («debug») дословно переводится как «избавление от насекомых» (bug – насекомое, жучок). Действительно, поиск ошибок в программе часто напоминает ловлю блох.

Термин «баг» к программному коду впервые был применен в 1946 г. Автор первого компилятора Грейс Хоппер работала в Гарвардском университете с вычислительной машиной Harvard Mark II. Проследив возникшую ошибку в работе программы до электромеханического реле машины, она нашла между замкнувшими контактами сгоревшего мотылька. Извлеченное насекомое было вклеено скотчем в технический дневник с сопроводительной иронической надписью: «Первый реальный случай обнаружения жучка».

Рассмотрим работу отладчика на примере программы деления двух чисел. Программа запрашивает делимое и делитель. Если делитель равен 0, то выводится сообщение о невозможности выполнить операцию. В противном случае выводится результат деления:

```
#include <iostream>
using namespace std;
void main()
{
	setlocale(LC_ALL,"rus");
	double a,b;
	printf("Введите делимое: ");
	scanf("%d",&a);
	printf("Введите делитель: ");
	scanf("%d",&b);
	if(b=0)
		printf("На 0 делить нельзя!\n");
	else
		printf("Результат деления= %5.2f\n",a/b);
	system("pause");
}
```

Программа содержит ошибки (хорошо, если заметили их сразу). Но, допустим, Вы пока не можете понять, почему программа выводит неверный результат (рисунок 4.9), ведь ожидали ответ 2.

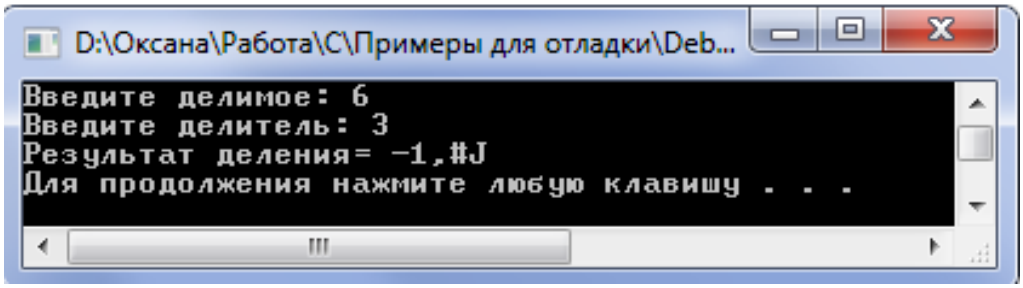


Рисунок 4.9 – **Результат работы программы с ошибками**

Попробуем найти эти ошибки с помощью отладки.

Для начала попытаемся остановить программу после ввода данных и проверить, правильно ли они были получены и размещены в памяти. Для этого щелкнем левой кнопкой мыши на серой полосе слева от программы напротив оператора if. Появится красный кружок – это *точка останова*, или *точка прерывания (Break Point)*. В процессе отладки программа остановится *перед* выполнением отмеченного оператора. Повторный щелчок по кружку снимает точку останова (рисунок 4.10).

Другой способ поставить точку останова – установить курсор на нужный оператор и нажать кнопку «F9».

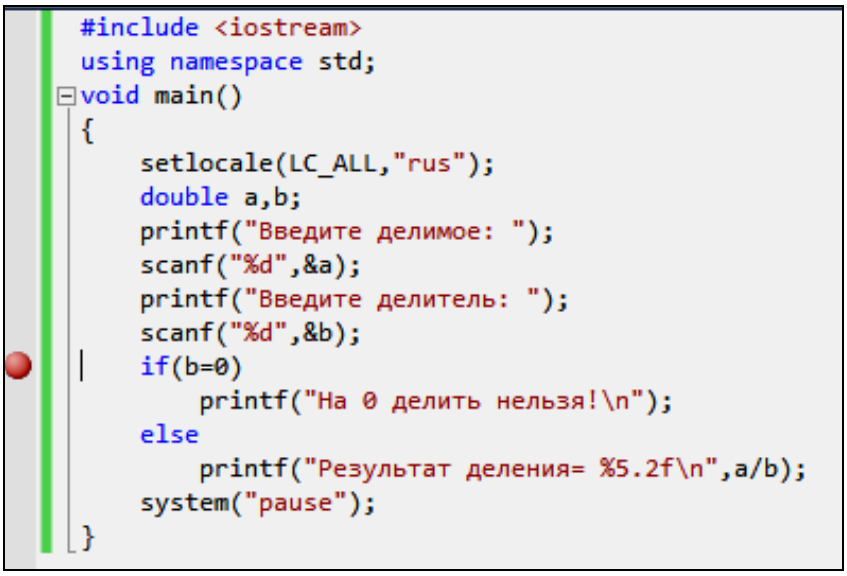


Рисунок 4.10 – **Точка останова**

Запускаем программу на выполнение обычным способом (щелчком по зеленому треугольнику на панели инструментов или нажав кнопку «F5»). Идет выполнение программы: запрашиваются значения делимого и делителя (вводим опять 6 и 3) и останавливаемся на точке прерывания. Причем текущий оператор (который будет выполняться следующим) отмечен желтой стрелкой (рисунок 4.11). При дальнейшем выполнении программы желтая стрелка может продвигаться дальше, а точка останова будет оставаться на своем месте, пока не будет снята.

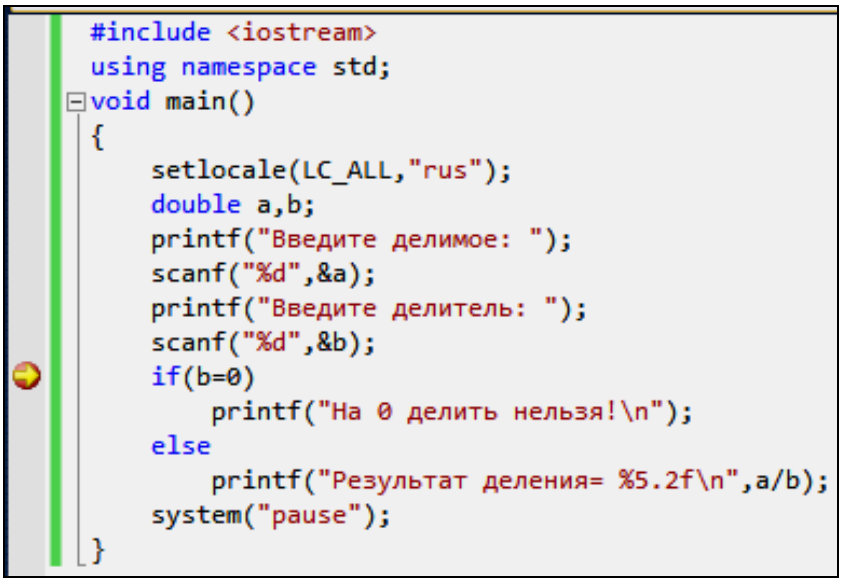


Рисунок 4.11 – **Точка останова и текущий оператор**

Теперь нужно проверить, правильно ли выполнен ввод. Для этого есть несколько *способов*:

• Навести курсор мышки на переменную – отобразится ее значение во всплывающем окне (рисунок 4.12).

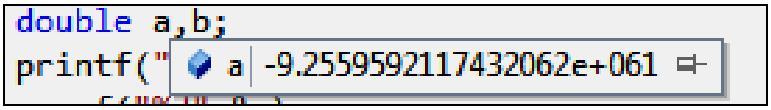


Рисунок 4.12 – **Всплывающее окно со значением переменной**

• Использовать окно локальных переменных (*Локальные, Locals*), которое выводится внизу рабочей области. Если это окно отсутствует, то его можно вывести на экран, выбрав *Отладка* → *Окна* → *Локальные*. Аналогично можно вывести (скрыть) и другие окна просмотра, о которых речь пойдет ниже.

В окне *Локальные* (рисунок 4.13) отражаются значения локальных переменных из текущего контекста. Отладчик заполняет это окно автоматически.

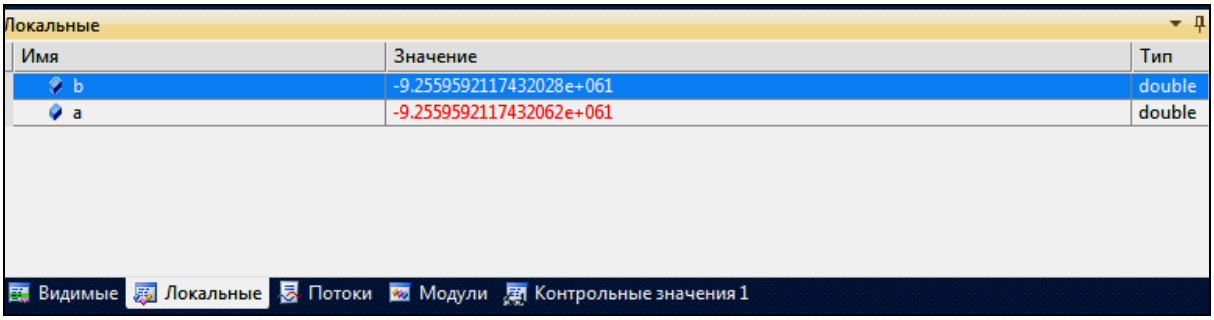


Рисунок 4.13 – **Окно локальных переменных**

• Использовать окно *Контрольные значения* (*Watch*). В это окно программист может сам добавлять переменные, которые нужно отслеживать. Кроме того, туда можно добавить и выражения, которые зачем-либо нужно рассчитать в процессе отладки. Вообще имеется четыре окна контрольных значений, но пока будем пользоваться одним.

Чтобы добавить переменную в это окно, наберем ее идентификатор в поле *Имя*. Второй способ – выделить имя в тексте программы и перетащить мышью в окно *Контрольные значения*. Этот способ особенно удобен, если переменная имеет длинное имя.

Введем в это окно переменные и выражение, результат которого желаем увидеть (рисунок 4.14).

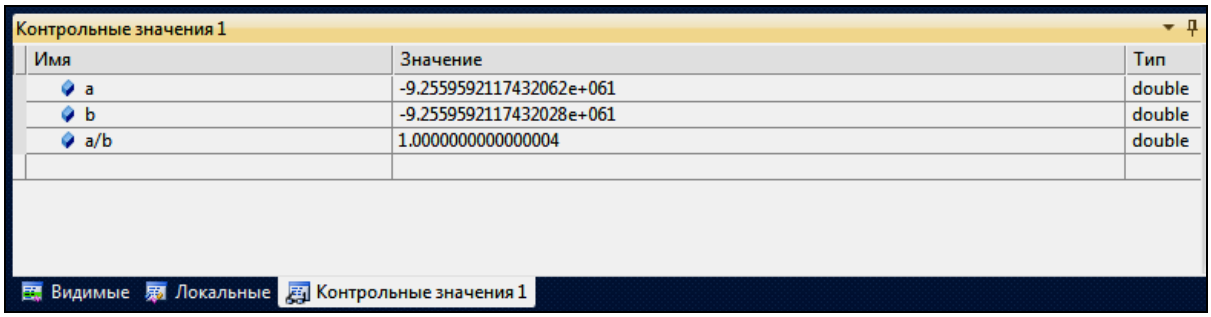


Рисунок 4.14 – **Окно *Контрольные значения***

• Использовать окно *Видимые* (*Autos*). Окно содержит значения переменных, которые были использованы в текущей и предыдущей строке кода. Как и окно *Локальные*, оно заполняется отладчиком автоматически (рисунок 4.15).

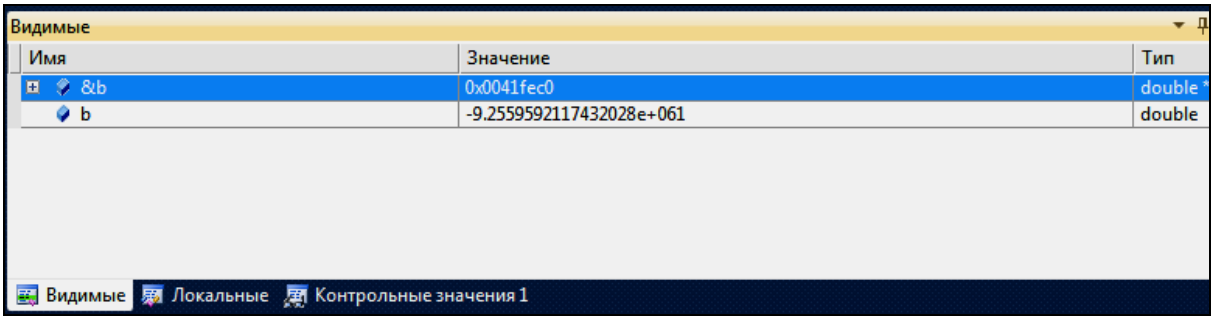


Рисунок 4.15 – **Окно *Видимые***

Любой из этих способов подтверждает, что данные введены неправильно (вводили 6 и 3). Следовательно, ошибку нужно искать в функции scanf. Конечно, не в реализации этой функции, а в ее вызове в программе. Проанализировав тип переменных *a* и *b* и формат ввода, обнаруживается ошибка: были введены вещественные числа по формату %d, предназначенному для целых. Необходимо исправить эту ошибку, поставив правильный формат: `scanf("%lf",&b);`.

Поскольку ошибка найдена и исправлена, следует перекомпилировать программу и запустить ее сначала. Удобный способ для этого – панель инструментов отладки.

Панель инструментов отладки

Эта панель инструментов появляется в окне Visual Studio только в процессе отладки (рисунки 4.16 и 4.17).

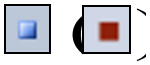
Рисунок 4.16 – **Панель инструментов «Отладка» в Visual Studio 2010**

Рисунок 4.17 – **Панель инструментов «Отладка» в Visual Studio 2017**

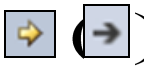
Рассмотрим кнопки на этой панели:

 – продолжить (F5). Если нажать эту кнопку, то выполнение программы продолжится в обычном режиме либо до конца, либо до следующей точки останова. Это такая же кнопка, как и на панели инструментов «Стандартная», которой пользовались для запуска программы.

 – пауза (Ctrl + Alt + Break). Эта кнопка прерывает выполнение программы на текущем операторе и переходит в режим ожидания указаний (она неактивна, поскольку в данный момент находимся в режиме паузы). Ее можно использовать в ситуациях, когда программа «зависает», чтобы выйти в режим отладки и проверить, какие именно операторы выполняются в бесконечном цикле.

 – остановить отладку (Shift + F5). Выполнение программы заканчивается досрочно. Например, в случае, если уже обнаружили ошибку, исправили ее и нужно текущее выполнение закончить и перейти к компиляции нового варианта программы.

 – перезапустить (Ctrl + Shift + F5). Эта кнопка заканчивает текущее выполнение и запускает программу заново (если код изменился, то он перекомпилируется перед новым запуском).

 – показать текущий оператор (впрочем, он и так показан желтой стрелкой в тексте программы).

 – шаг с заходом (F11). Использование этой кнопки приводит к выполнению одного текущего оператора (отмеченного желтой стрелкой). Причем, если этот оператор содержит вызов функции, то процесс пошагового выполнения «заходит» в эту функцию, показывая последовательность выполнения операторов внутри ее кода. Например, если стрелка стояла на функции printf, то нажав эту кнопку, увидим окно с кодом функции printf, причем стрелка стоит в самом начале (рисунок 4.18). Конечно, все стандартные функции уже давно отлажены, и нет смысла заходить внутрь их. Шаг с заходом нужно будет использовать тогда, когда начнете создавать свои функции.

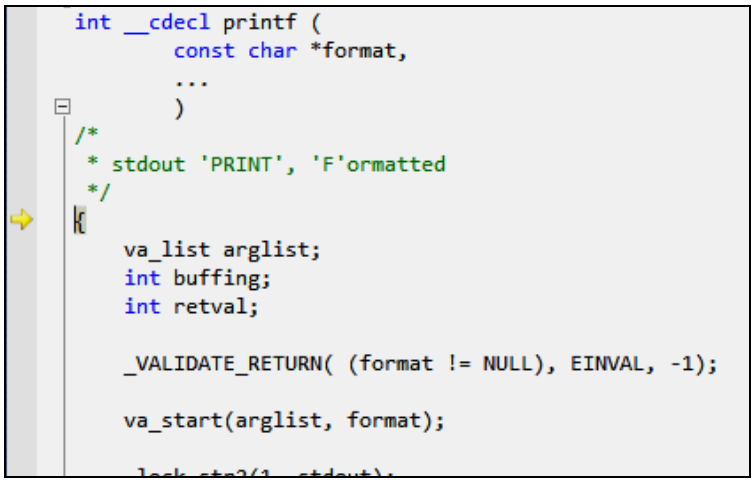


Рисунок 4.18 – **Шаг с заходом в функцию printf()**

 – шаг с обходом (F10). Выполняется один текущий оператор. Но если этот оператор содержит вызов функции, то в нее процесс не «заходит», а выполняет как один шаг.

 – шаг с выходом (Shift + F11). Выполняются все операторы до конца функции, возвращается управление в вызвавшую функцию и происходит переход в режим паузы. Можно использовать эту кнопку, если случайно зашли внутрь стандартной функции и нужно вернуться в свою программу.

(Hex). Эта кнопка позволяет задать (снять) вывод данных в окнах в шестнадцатеричном формате (для целых).

Нажмем кнопку  (рестарт) либо последовательно кнопки  (остановить) и  (продолжить). Текущее выполнение закончится досрочно, код будет откомпилирован, и начнется выполнение программы заново. Останов произойдет опять на точке прерывания после ввода данных. Анализ значений переменных убеждает нас в том, что на этот раз данные введены верно и результатом работы программы должно быть число 2 (рисунок 4.19).

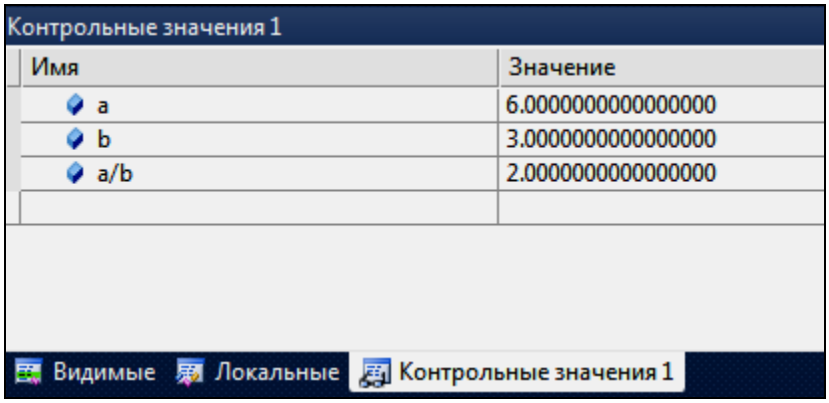


Рисунок 4.19 – **Значения переменных после ввода данных**

Для дальнейшей отладки будем выполнять пошагово данную программу, без захода внутрь функций (которые у нас все стандартные). Нажимаем F10 (или кнопку ). Следующий оператор, который должен быть выполнен – вывод по ветке else (рисунок 4.20). Это верно, так как ввели $b = 3$.

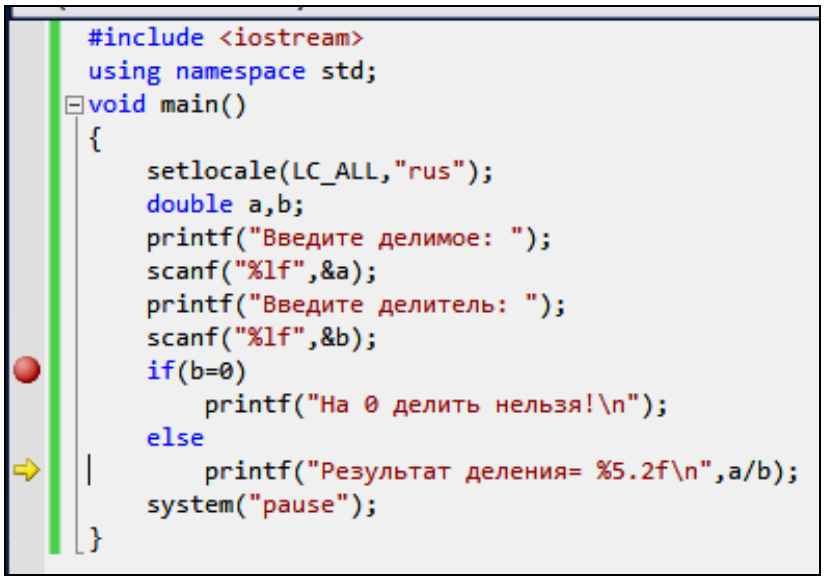


Рисунок 4.20 – **Результат одного шага после останова**

Однако если изучить состояние переменных в окне *Контрольные значения* (рисунок 4.21), то обнаружим, что значение делителя *b* изменилось. Красным цветом в этом окне выделяются те значения, которые изменились на предыдущем шаге.

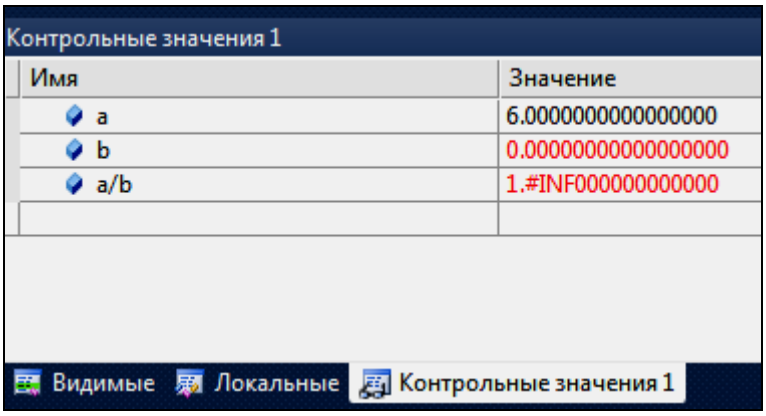


Рисунок 4.21 – **Значение переменных после одного шага отладки**

Таким образом, переменная *b* получила значение 0 во время выполнения оператора if. Это значит, что вместо *сравнения b* с нулем ему *присвоили* 0.

Теперь можно исправить и эту ошибку в программе:

```
if(b==0)
…
```

и продолжить отладку. Поскольку ошибок больше нет, получим верный результат теста (рисунок 4.22).

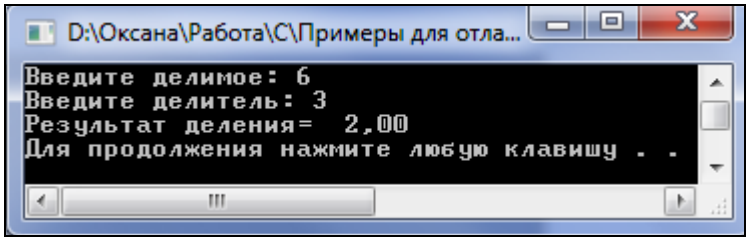


Рисунок 4.22 – **Результат работы отлаженной программы**

Точка останова

Для точки останова можно задать условие, при выполнении которого она будет срабатывать. Например, дана программа расчета факториала числа *n* (факториал – это произведение всех чисел от 1 до *n* включительно: $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$) (рисунок 4.23). Эта программа работает неправильно. Например, для $n = 5$ она должна давать результат 120, а дает 24. Внутрь цикла поставим точку прерывания и начнем проверять работу цикла. Видим, что вначале цикл работает правильно. Поэтому нужно пропустить первые итерации и остановиться, когда переменная цикла $i == 5$.

```
#include <iostream>
using namespace std;
void main()
{
    setlocale(LC_ALL,"rus");
    int n,res=1;
    printf("Введите число: ");
    scanf("%d",&n);
    for(int i=1;i<n;i++)
        res*=i;
    printf("%d!=%d\n",n,res);
    system("pause");
}
```

Рисунок 4.23 – **Программа вычисления факториала**

Для задания условия останова вызовем контекстное меню точки прерывания (правой кнопкой мыши по точке) и выберем «условие» (рисунок 4.24). Зададим условие $i == 5$ (рисунок 4.25). Форма точки прерывания изменится: на ней появится значок «+».

Запускаем программу и задаем $n = 5$. Программа отработает до конца, и останова не произойдет. Это говорит о том, что не происходит умножения накапливаемого произведения на последнее число: у нас неверно задано условие выхода из цикла. Исправляем:

```
for(int i=1;i<=n;i++)
     res*=i;
```

Программа теперь работает верно, а «умная» точка останова избавила нас от утомительного «прокручивания» цикла до того места, где локализована ошибка.

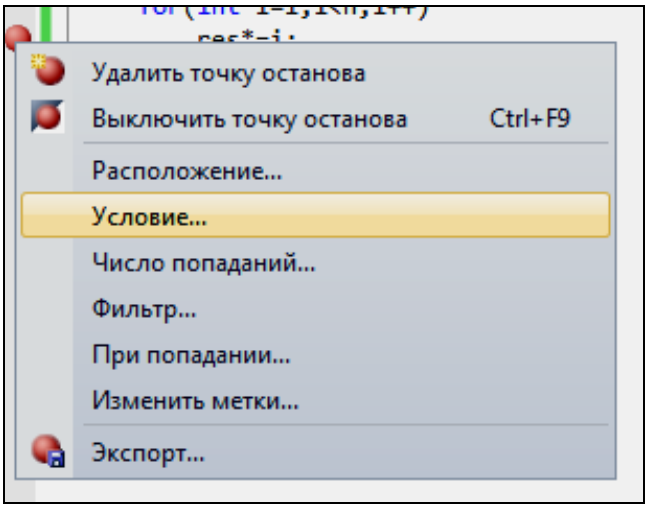


Рисунок 4.24 – **Контекстное меню точки останова**

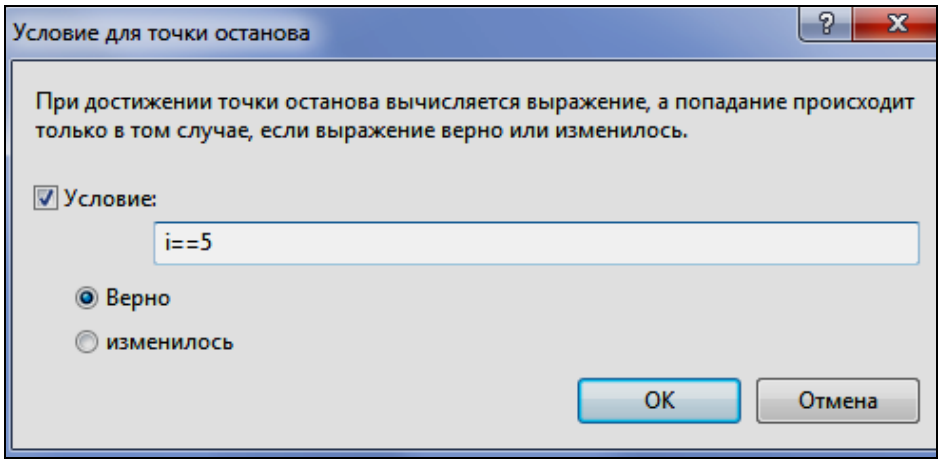


Рисунок 4.25 – **Условие для точки останова**

В заключение приведем небольшой справочник по использованию горячих клавиш в отладчике (таблица 4.1).

Таблица 4.1 – **Горячие клавиши отладчика Visual Studio**

Сочетание клавиш	Назначение
F9 (или щелчок в левом поле)	Поставить (убрать) точку останова
F10	Пошаговое выполнение без захода в функцию
F11	Пошаговое выполнение с заходом в функцию
Ctrl + F10	Начать пошаговое выполнение с определенной строки
F5	Начать (продолжить) выполнение программы в обычном режиме
Shift + F5	Прервать работу программы
Ctrl + Shift + F5	Перезапустить

4.9. Вложенные циклы

Так же как и другие алгоритмические конструкции циклы можно «вкладывать» друг в друга. Внутренний цикл в тексте программы сдвигается вправо на 3–4 позиции. Понимать и отлаживать вложенные циклы сложнее, чем вложенные операторы ветвления.

Рассмотрим сначала пример вложенных циклов, которые не зависят друг от друга.

Пример 4.8. Необходимо вывести на экран прямоугольник из звездочек (рисунок 4.26). Ширина и высота прямоугольника запрашивается у пользователя.

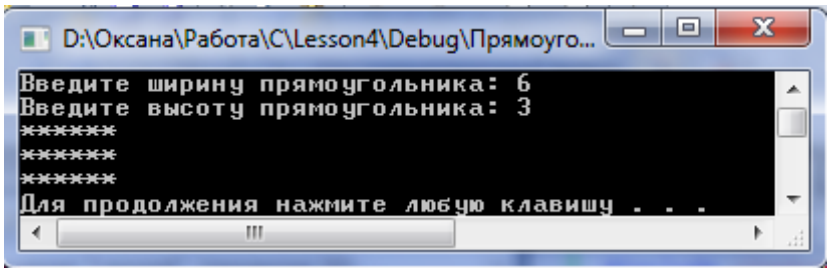


Рисунок 4.26 – **Результат работы программы «Прямоугольник»**

Введем ширину прямоугольника в переменную *len*, а высоту – в переменную *h*.

В процессе работы программы будем по очереди выводить строки, составляющие прямоугольник. Поскольку высота прямоугольника *h*, действие по выводу строки будет выполняться *h* раз. Для этого удобно использовать цикл for:

```
for(int i=1;i<=h;i++)
{
     //Вывод строки i;
}
```

Здесь *i* – номер строки, который меняется от 1 до *h* с шагом 1.

Вывод каждой строки состоит в выводе *len* звездочек. Потом курсор переводится на новую строку:

```
//вывод одной строки:
for(int j=1;j<=len;j++)
     cout<<'*';
cout<<"\n";
```

Обратите внимание, что вывод символа «\n» выполняется не в цикле, а после него. Поэтому этот оператор находится на том же уровне, что и цикл for. Переменная цикла *j* означает номер звездочки в строке.

Теперь соберем эти фрагменты в единый код:

```
#include <iostream>
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     int h,len;
     cout<<"Введите ширину прямоугольника: ";
     cin>>len;
     cout<<"Введите высоту прямоугольника: ";
     cin>>h;
     for(int i=1;i<=h;i++)
     {
          for(int j=1;j<=len;j++)
               cout<<'*';
          cout<<"\n";
     }
     system("pause");
}
```

Внутренний цикл будет выполняться от начала и до конца для каждого значения *i*. Таким образом, будет выведено *len* · *h* звездочек. Последовательность изменения переменных цикла (для *len* = 6, *h* = 3) показана в таблице 4.2. Для лучшего понимания программы выполните ее в пошаговом режиме отладчика.

Таблица 4.2 – **Значения переменных цикла в процессе выполнения программы «Прямоугольник»**

i	1	2	3
j	1, 2, 3, 4, 5, 6, перевод курсора	1, 2, 3, 4, 5, 6, перевод курсора	1, 2, 3, 4, 5, 6, перевод курсора

Рассмотрим теперь пример, когда внутренний цикл зависит от параметра внешнего цикла.

Пример 4.9. Необходимо вывести на экран треугольник из звездочек. Высота треугольника запрашивается у пользователя (рисунок 4.27).

Принцип работы программы будет тот же. Организуем цикл вывода строк:

```
for(int i=1;i<=h;i++)
{
     //Вывод строки i;
}
```

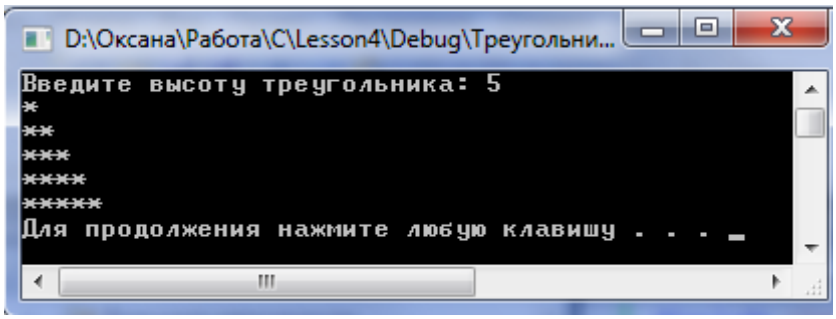


Рисунок 4.27 – **Результат работы программы «Треугольник»**

В этом случае число звездочек в строке будет зависеть от номера строки: в первой строке – 1 звездочка, во второй строке – 2 и т. д. В общем случае в строке номер *i* нужно вывести *i* звездочек, а потом перевести курсор на новую строку:

```
//вывод i-ой строки
for(int j=1;j<=i;j++)
     cout<<'*';
cout<<"\n";
```

В этом фрагменте специально выделено условие выхода из цикла: *j* <= *i*. Таким образом, для высоты треугольника в 5 символов будет выведено (1 + 2 + 3 + 4 + 5), т. е. 15 звездочек. Полный код программы имеет вид:

```
#include <iostream>
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     int h;
     cout<<"Введите высоту треугольника: ";
     cin>>h;
     for(int i=1;i<=h;i++)
     {
          for(int j=1;j<=i;j++)
               cout<<'*';
          cout<<"\n";
     }
     system("pause");
}
```

Значения переменных цикла для примера *h* = 5 показаны в таблице 4.3. Пройдите по этой программе отладчиком и убедитесь, что полностью понимаете процесс выполнения кода.

Таблица 4.3 – **Значения переменных цикла в процессе выполнения программы «Треугольник»**

i	1	2	3	4	5
j	1, перевод курсора	1, 2, перевод курсора	1, 2, 3, перевод курсора	1, 2, 3, 4, перевод курсора	1, 2, 3, 4, 5, перевод курсора

4.10. Структурное программирование

Метод структурного программирования возник в 70-е гг. XX в. Идея была предложена Эдсгером Дейкстра, а развил ее Никлаус Вирт. Это время характеризовалось ростом сложности программ, что сопровождалось большими затратами времени на их разработку и отладку. Э. Дейкстра выдвинул идею, что во всем виноват оператор goto. Возможность передачи управления в любую точку программы приводит к сложному для понимания алгоритму, к потерям памяти и появлению сбойных участков алгоритмов (например, управление передается в точку,

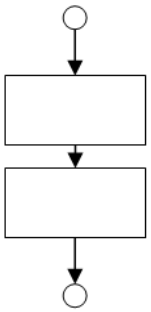
где еще не инициализированы нужные переменные). Поэтому ключевой идеей структурного программирования стал отказ от оператора goto и использование только трех типов управляющих структур:

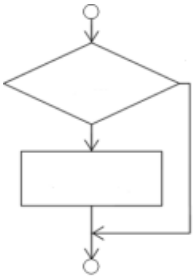
- следование;
- выбор;
- повторение (цикл).

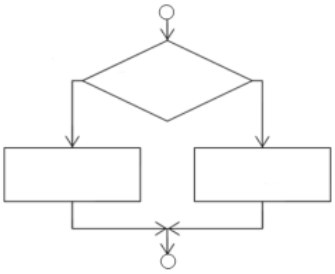
При этом в языке С применяется три типа конструкции выбора: if (выбор с одной альтернативой), if-else (выбор с двумя альтернативами) и switch (множественный выбор). А также используется три вида циклов: while (цикл с предусловием), do-while (цикл с постусловием) и for (цикл с параметром). Итого семь управляющих конструкций (их блок-схемы приведены в таблице 4.4). Другие конструкции не разрешены.

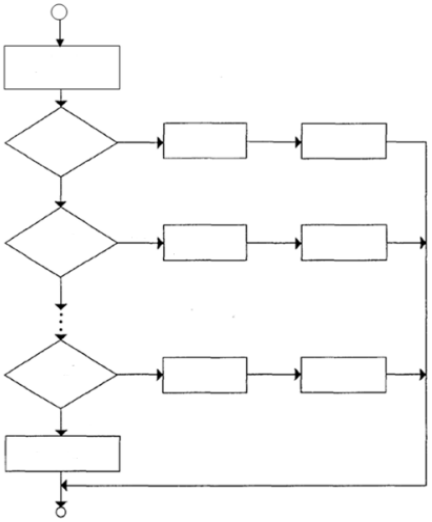
Таблица 4.4 – **Алгоритмические конструкции языка программирования**

Алгоритмическая конструкция	Оператор языка С

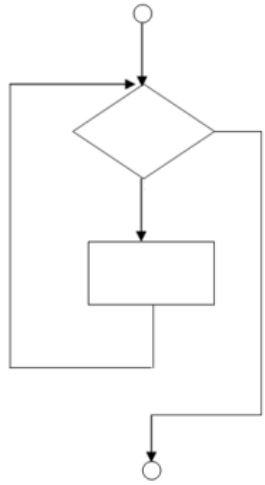
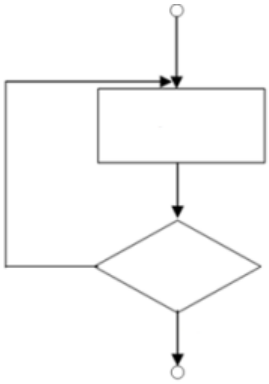
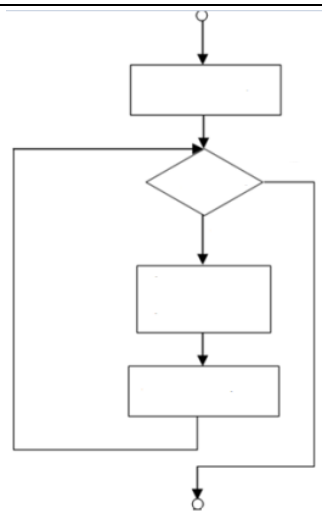
	{
	оператор 1;
	оператор 2;
	}

	if (условие) оператор;

	if (условие)
	оператор 1;
	else
	оператор 2;

	switch (выражение)
	{
	case значение1: оператор 1; break;
	case значение 2: оператор 2; break;
	…
	default: оператор;
	}

Окончание таблицы 4.4

Алгоритмическая конструкция	Оператор языка С
	while (условие) оператор;
	do оператор; while (условие);
	for (инициализация; условие; модификация) оператор;

Важно, что каждая такая алгоритмическая конструкция имеет только один вход и один выход. Эти точки на блок-схемах изображаются в виде кружков. Соединения конструкций в программе реализуется только через эти точки входа и выхода. Причем разрешены всего два способа объединения структур:

- следование;
- вложение.

В случае следования точка выхода одной структуры соединяется со входом другой.

В случае вложения действует правило, позволяющее любой прямоугольник блок-схемы (действие) заменить другой структурой.

Пример применения этого правила показан на рисунке 4.28.

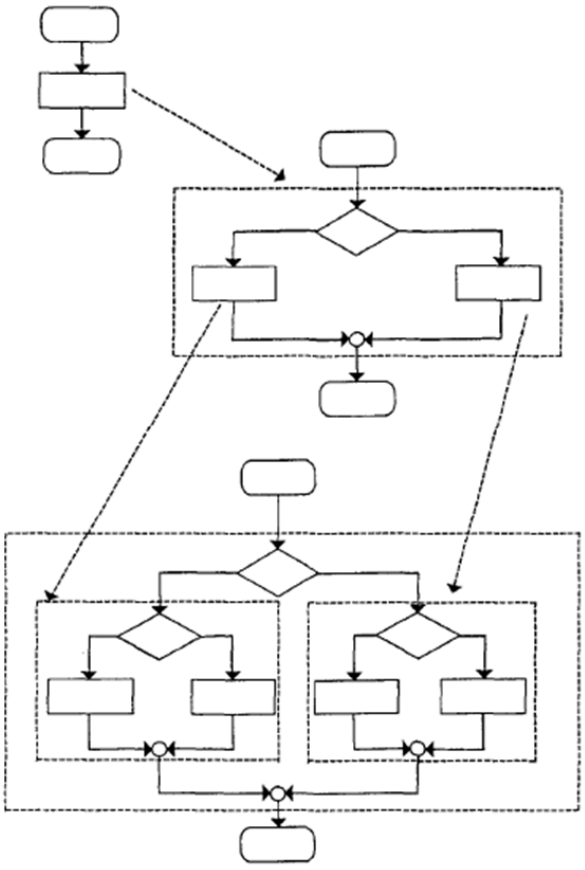


Рисунок 4.28 – **Пример применения правила вложения**

При условии соблюдения этих правил запутанная, неструктурная программа просто не может быть разработана. Например, не может быть получен алгоритм, показанный на рисунке 4.29. Алгоритмические конструкции не могут частично пересекаться, а должны быть полностью вложены друг в друга. В коде программы степень вложения одной конструкции в другую отмечается сдвигом вправо на 3–4 позиции.

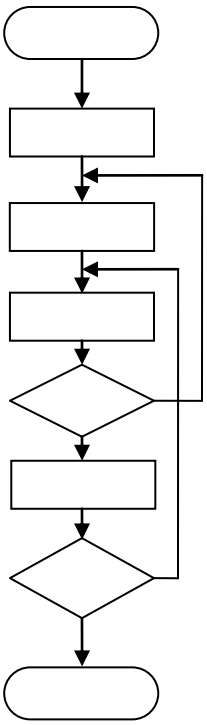


Рисунок 4.29 – **Неструктурный алгоритм**

4.11. Пошаговое нисходящее проектирование программ

Нисходящее проектирование программ, т. е. подход к разработке «сверху вниз», также считается одним из основных принципов структурного программирования.

Суть этого подхода заключается в том, что сначала формулируются самые верхние, общие шаги алгоритма на псевдокоде. Затем происходит постепенная детализация этих шагов до тех пор, пока они не смогут быть полностью реализованы на языке программирования.

Основная идея этого метода – не пытаться программировать сразу. Пошаговая детализация автоматически заставляет программиста формировать понятную ему же структуру программы. Программист при этом хорошо представляет себе работу каждой конкретной подзадачи, ее входные и выходные данные и потому в состоянии правильно ее протестировать. Также упрощается и последующая отладка – необходимо проверить только, верно или неверно отработала очередная подзадача. Круг поиска ошибок сужается до одного небольшого фрагмента кода.

Применение метода нисходящего проектирования рассмотрим на следующем примере.

Пример 4.10. Программа обобщает результаты сдачи экзамена для группы из десяти студентов. Пользователь вводит оценку за экзамен по 10-балльной системе. Экзамен считается сданным, если оценка составляет 4 балла и выше. Нужно подсчитать количество сдавших и несдавших экзамен студентов и вывести сводку на экран. Если экзамен сдали более восьми студентов, вывести сообщение «Повысить плату за курс».

Решение начинается с формулирования самой общей задачи программы, или задачи ***первого уровня***: *проанализировать результаты экзамена и решить, должна ли быть повышена плата за курс*.

Затем выделяем подзадачи ***второго уровня***, которые отметят основные этапы решения:

1. Инициализировать переменные.

2. Ввести десять оценок и подсчитать число сданных и несданных экзаменов.

3. Вывести сводку результатов и решить, должна ли быть повышена плата за курс.

Для перехода к следующему уровню детализации нужно подумать о том, какие переменные потребуются в программе (можно сразу же назначить им имена). Очевидно, будут нужны:

- счетчик студентов – student;
- счетчик числа сдавших экзамен – passed;
- счетчик числа несдавших экзамен – failed;
- текущая оценка – result.

Счетчик студентов – это номер очередного студента, данные которого вводят. Поэтому он должен быть инициализирован единицей. Счетчики числа сдавших и несдавших студентов инициализируем нулем, так как они служат для накопления данных в процессе работы программы. Текущую оценку можно не инициализировать, так как при вводе в нее все равно предыдущее значение разрушится. Поэтому этап инициализации переменных (*этап 1*) можно детализировать на ***третьем уровне*** следующими действиями:

```
student=1; //номер студента;
passed=0; //счетчик сдавших;
failed=0; //счетчик несдавших;
```

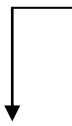
Отметим, что иногда (в зависимости от сложности программы) детализацию этапа инициализации переменных можно отложить и выполнить сначала детализацию основного этапа алгоритма. В процессе обдумывания основного этапа выяснится, какие именно переменные понадобятся и как их нужно инициализировать. Тогда можно вернуться к этапу 1 и реализовать необходимые начальные присваивания.

Вообще, нужно стараться следовать естественной логике событий: если не видите необходимости детализировать какой-то фрагмент алгоритма именно сейчас – отложите его. Один из шуточных принципов программирования: «Не откладывай на завтра то, что можно отложить на послезавтра».

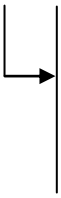
Перейдем к детализации *этапа 2*. Очевидно, что ввод данных нужно выполнять 10 раз, т. е. пока номер очередного студента меньше или равен 10.

Алгоритм ***третьего уровня*** будет выглядеть так:

Пока *student<=10*
нц
Ввести и обработать результаты следующего студента;
student++; //увеличить номер студента
кц



В этом псевдокоде в дальнейшей детализации нуждается фраза «Ввести и обработать…», поэтому уточним, что под этим понимается (***четвертый уровень***).



Ввести результат следующего студента в переменную result;
Если *result>=4*
passed++; //увеличить счетчик сдавших
иначе
failed++; //увеличить счетчик несдавших
все-если

Какой именно фрагмент алгоритма подвергся дальнейшей детализации, можно указывать с помощью стрелок, как показано выше.

Перейдем теперь к детализации *этапа 3* (вывод результатов):

Вывести количество сдавших экзамен (passed);
Вывести количество несдавших экзамен (failed);
Если *passed>8 //количество сдавших>8*
Вывести «Повысить плату за курс»
Все-если

Теперь осталось собрать фрагменты алгоритма, «вставив» их на нужное место в псевдокоде. В результате получим полный алгоритм программы:

Начало
student=1; //номер студента
passed=0; //счетчик сдавших
failed=0; //счетчик несдавших
Пока *student<=10*
нц
Ввести результат следующего студента в переменную result;
Если *result>=4*
passed++; //увеличить счетчик сдавших
иначе
failed++; //увеличить счетчик несдавших
все-если
student++; //увеличить номер студента
кц
Вывести количество сдавших экзамен (passed);
Вывести количество несдавших экзамен (failed);
Если *passed>8 //количество сдавших>8*
Вывести «Повысить плату за курс»
Все-если
Конец

При этом задачи, которые были сформулированы на первом этапе, можно оставить в виде комментариев (а можно и не оставлять, как это сделано в примере). Полученную программу уже совершенно легко перевести в код на языке С. Этапы алгоритма в тексте программы принято отделять пустой строкой. Код программы следующий:

```
//проанализировать результаты экзамена
#include <iostream>
using namespace std;
```

```
void main()
{
     setlocale(LC_ALL,"rus");

//инициализировать переменные
     int student=1; //номер студента
     int passed=0, failed=0; //число сдавших и несдавших
     int result; //очередная оценка

//Ввести десять оценок и подсчитать количество сдавших и несдавших
     while(student<=10)
     {
          // Ввести и проанализировать следующую оценку
          cout<<"Введите следующую оценку: ";
          cin>>result;
          if(result>=4)
               passed++;
          else
               failed++;
          student++;
     }

//Вывести сводку результатов экзамена и определить, должна ли быть
//повышена плата за курс
     cout<<"Сдали экзамен "<<passed<<" чел\n";
     cout<<"Не сдали экзамен "<<failed<<" чел\n";
     if(passed>8)
          cout<<"Повысить плату за курс\n";
     system("pause");
}
```

Задачи

Простые циклы

Задача 4.1. Найти произведение всех целых нечетных чисел в диапазоне, указанном пользователем.

Задача 4.2. Построить таблицу значений функции $y(x)$:

$$y=\begin{cases}2, & \text{при } x\le -4;\\ -\dfrac{1}{2}x & \text{при } -4<x\le 2;\\ x-3 & \text{при } x>2.\end{cases}$$

Границы диапазона и шаг табуляции запросить у пользователя.

Задача 4.3. Пользователь вводит два целых числа. Необходимо вывести все целые числа, на которые оба введенных числа делятся без остатка.

Задача 4.4. Программа запрашивает два целых числа x и y, после чего вычисляет и выводит значение x в степени y (не использовать функцию pow()).

Задача 4.5. Найти максимум из n чисел, которые последовательно вводятся с клавиатуры. Программа сначала запрашивает количество чисел (n), а потом сами числа.

Задача 4.6. Разработать программу, которая выводит на экран горизонтальную или вертикальную линию из символов. Число символов, какой будет символ и какая будет линия (вертикальная или горизонтальная), указывает пользователь.

Для самостоятельного решения

Задача 4.7. Пользователь вводит с клавиатуры целое неотрицательное число. Программа должна выполнить следующие задачи:

- определить, сколько в данном числе цифр;
- определить сумму цифр;
- вывести цифры числа в обратном порядке.

Задача 4.8. Разработать программу "Конвертор валют". Программа выводит меню, предлагая выбрать вариант конверсии (BYR → USD, BYR → EUR, BYR → RUB и т. д.). После выполнения конверсии одной суммы программа предлагает выбрать снова и т. д. Одним из элементов меню является символ «q» – выход из программы. При выборе этого элемента программа заканчивает работу.

Задача 4.9. Написать игру «Угадай число». Программа загадывает число в диапазоне от 1 до 500. Пользователь пытается его угадать. После каждой попытки программа выдает подсказки, больше или меньше его число загаданного. В конце программа выдает статистику: за сколько попыток угадано число. Предусмотреть выход по 0 в случае, если пользователю надоело угадывать число.

Подсказка: для генерации в программе случайного числа подключите библиотеку `#include <time.h>` и используйте следующий код:

```
srand(time(0));
int my=rand()%500 +1;
```

Тогда в переменной *my* будет содержаться загаданное компьютером число.

Задача 4.10. Пользователь вводит число. Определить, является ли оно простым. Как известно, число называется простым, если оно делится только на себя и на единицу.

Задача 4.11. Написать программу, которая проверяет пользователя на знание таблицы умножения. Программа выводит на экран два числа, пользователь должен ввести их произведение. Разработать несколько вариантов (должны отличаться сложностью и количеством вопросов). Вывести оценку знаний пользователю.

Задача 4.12. Вывести на экран числа от 100 до 999 с заданным шагом в заданное количество столбиков.

Задача 4.13. Вывести на экран все ASCII-символы и их коды.

Задача 4.14. Для двух чисел определить наименьшее общее кратное.

Вложенные циклы

Задача 4.15. Написать программу, которая выводит на экран простые числа в диапазоне от 2 до 1 000.

Задача 4.16. Написать программу, которая выводит на экран фигуру вида

```
**********
*        *
*        *
**********
```

Ширина и высота фигуры задаются пользователем.

Задача 4.17. Ввести высоту треугольника и вывести на экран треугольник следующего вида:

```
   *
  **
 ***
****
```

Задача 4.18. Ввод последовательности чисел заканчивается 0. Написать программу подсчета суммы тех чисел, порядковые номера которых являются простыми числами.

Задача 4.19. Ввести ширину треугольника и вывести на экран треугольник вида

```
*****
 ***
  *
```

Для самостоятельного решения

Задача 4.20. Вывести на экран шахматную доску 8 × 8 клеток с размером клеток *n* символов. При этом белые клетки выводятся символом «_», а черные – символом «*». Например, фрагмент доски может выглядеть так:

```
***___***___***___***___
***___***___***___***___
***___***___***___***___
___***___***___***___***
___***___***___***___***
___***___***___***___***
```

Задача 4.21. Когда в комнате развелось уже *n* мух, Иван Петрович открыл квартиру и, размахивая полотенцем, начал выгонять их на улицу. Чтобы выгнать одну муху, он затрачивал 1 мин, но через каждые 5 мин в комнату залетала новая муха. Когда в комнате стало меньше, чем 10% от начального количества мух, процесс борьбы с мухами ускорился вдвое. Определить, сколько мух осталось в комнате через *k* минут? Через сколько минут Иван Петрович остался в комнате один?

Задача 4.22. В введенном промежутке натуральных чисел найти те, количество делителей у которых не меньше введенного значения. Для найденных чисел вывести на экран количество делителей и все делители.

Задача 4.23. Вывести на экран, из каких простых множителей состоит введенное натуральное число.

Задача 4.24. Найти все совершенные числа до 10 000. Совершенное число – это такое число, которое равно сумме всех своих делителей, кроме себя самого. Например, число 6 является совершенным, так как кроме себя самого делится на числа 1, 2 и 3, которые в сумме дают 6.

Задача 4.25. Среди натуральных чисел, которые были введены, найти наибольшее по сумме цифр. Вывести на экран это число и сумму его цифр.

Задача 4.26. Вывести на экран таблицу умножения (от 1 до 9).

ТЕМА 5. МАССИВЫ

5.1. Объявление и инициализация массива

Массив – это структурированный тип данных, который представляет собой набор данных одного типа. Ко всему массиву целиком можно обращаться по имени. Чтобы обратиться к элементу массива, нужно указать имя массива и индекс (порядковый номер) в квадратных скобках, например, *a*[3].

Нумерация элементов в массиве начинается с 0. Таким образом, если, например, объявлен массив из десяти элементов, то используются индексы от 0 до 9.

Массивы бывают одномерные (с одним индексом) и многомерные (два и более индексов). Рассмотрим сначала одномерные массивы.

При объявлении *одномерного массива* указывается тип элемента, затем имя массива и число элементов в квадратных скобках. Имя массива должно подчиняться общим правилам создания индентификаторов в языке С. Число элементов – это целочисленное константное значение, т. е. на этом месте может быть целочисленный литерал, константа или выражение, значением которого является константа. Другими словами, на этапе компиляции число элементов массива должно быть однозначно известно, например:

```
double x[3]; //массив из трех вещественных элементов
const int SIZE=5;
char c[SIZE]; //массив из пяти элементов char;
char s[SIZE+2]; //массив из семи элементов char
```

Для задания размерности массива можно также использовать константу, описанную командой препроцессора #define:

```
#define N 10
…
int mas[N]; //массив из десяти элементов
```

Использование констант при объявлении массива является признаком хорошего стиля программирования. В этом случае при необходимости изменить количество элементов массива не нужно просматривать и изменять всю программу – достаточно изменить значение константы в одном месте.

Элементы массива располагаются подряд друг за другом, как показано на рисунке 5.1.

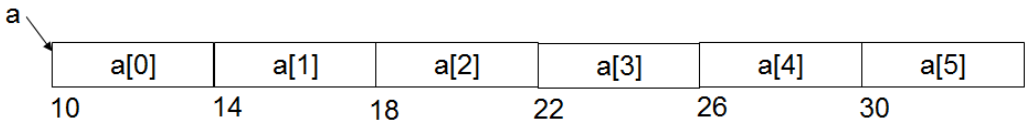


Рисунок 5.1 – **Размещение элементов массива в памяти**

Каждый элемент занимает столько памяти, сколько отводится под соответствующий тип данных. Например, если объявлен массив целых чисел из шести элементов (int *a*[6];), то каждый элемент занимает 4 байта. Если, например, массив расположен в памяти, начиная с адреса 10, то это адрес нулевого элемента. Элемент номер 1 расположен по адресу 10 + 4 = 14, а элемент номер 2 – по адресу 14 + 4 = 18 и т. д.

Имя массива – это фактически адрес начала массива (адрес нулевого элемента), т. е. значение *a* и &*a*[0] – одинаковые.

Общее число байт, занятое под массив, можно рассчитать, пользуясь функцией sizeof(тип данных), которая выдает количество байт, отводимых под переменную соответствующего типа. Например, sizeof(int) дает 4, а sizeof(double) дает 8. Таким образом,

Число байт, занятых под массив = Число элементов · sizeof (тип данных).

В нашем примере (6 · 4), т. е. 24 байта занимает весь массив.

Напомним, что параметром операции sizeof() может быть не только название типа, но и имя переменной, например:

```
int k, b[5];
cout <<sizeof(int) <<"\n"; //выводит 4;
cout <<sizeof(k) <<"\n"; //тоже выводит 4;
cout <<sizeof(b) <<"\n"; //выводит 20, т.е. 5*4
```

При объявлении можно сразу инициализировать элементы массива, если указать знак «=», и список значений элементов в фигурных скобках.

Если при этом не указывать количество элементов, то оно будет определено автоматически по количеству инициализирующих значений:

```
int a[]={3,1,5,6}; //объявление и инициализация массива из 4 элементов
```

Если же задать меньше значений, чем размер массива, то будут инициализированы первые значения, а остальные получат значение 0:

```
int b[5]={3,4,2}; //элементы получили значения 3,4,2,0,0
```

Если же инициализирующих значений больше, чем указанная размерность массива, то возникнет ошибка на этапе компиляции:

```
//double f[3]={0.1,0.3,0.4,0.5}; //ошибка
```

5.2. Ввод и вывод одномерного массива

Для работы с массивами обычно применяются циклы. Рассмотрим сначала алгоритмы ввода и вывода элементов массива.

Чтобы ввести значения всех элементов массива, используют цикл for (так как индекс – это целое число, он вполне может выступать в роли параметра цикла). Для удобства перед вводом очередного элемента будем выводить приглашение с его номером:

```
const int SIZE=6;
int a[SIZE];
cout<<"введите элементы массива \n";
for(int i=0;i<SIZE;i++)
{
     cout<<"a["<<i<<"]=";
     cin>>a[i];
     cout<<"\n";
}
```

Следует заметить, что индекс изменяется от 0 до SIZE – 1, поскольку элемента с номером SIZE не существует.

Выводить элементы одномерного массива удобно в строку (если, конечно, этих элементов не очень много). После вывода очередного элемента выводят знак табуляции, а после вывода всех элементов переводят курсор на новую строку:

```
cout<<"исходный массив:\n";
for(int i=0;i<SIZE;i++)
    cout<<a[i]<<"\t";
cout<<"\n";
```

5.3. Типовые алгоритмы работы с массивами

Сумма элементов массива

Переменная для накопления суммы инициализируется нулем. Затем перебирают все элементы массива и прибавляют каждый из них к сумме. После завершения прохода по массиву получают сумму всех элементов в этой переменной:

```
int sum=0;
for(int i=0;i<SIZE;i++)
     sum=sum+a[i];
cout<<"Сумма элементов массива = "<<sum<<"\n";
```

Блок-схема данного алгоритма показана на рисунке 5.2.

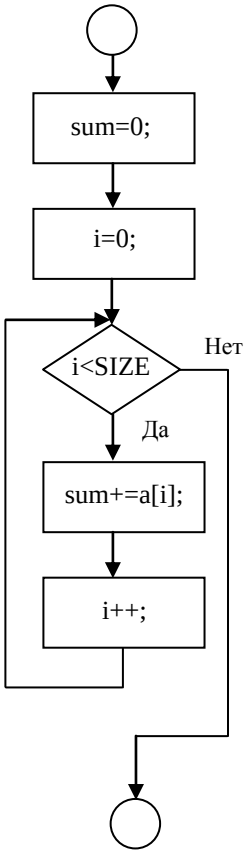


Рисунок 5.2 – **Блок-схема суммы элементов массива**

Поиск максимального значения в массиве

Для нахождения максимального значения в массиве вводят переменную *max*, в которой будет храниться максимум из просмотренных элементов. Сначала в эту переменную записывают нулевой элемент массива. Потом просматривают все элементы, начиная с первого, и сравнивают их с текущим максимумом. Если очередной элемент больше, максимум получает новое значение (старое значение при этом теряется). Если же текущий элемент меньше или равен предыдущему, то значение переменной *max* не изменяется:

```
int max=a[0];
for(int i=1;i<SIZE;i++)
     if(a[i]>max) max=a[i];
cout<<"Максимальное значение= "<<max<<"\n";
```

Блок-схема этого алгоритма показана на рисунке 5.3.

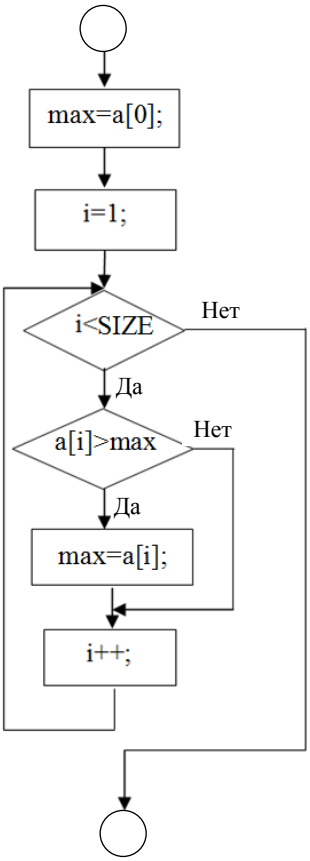


Рисунок 5.3 – **Блок-схема определения максимума в массиве**

Иногда бывает необходимо найти не только само максимальное значение, но и его место в массиве (т. е. индекс максимального элемента). Для того чтобы запомнить индекс максимума, вводят еще одну переменную *imax*, которая будет изменяться одновременно с максимумом:

```
max=a[0];
int imax=0; //текущий максимум на месте 0
for(int i=1;i<SIZE;i++)
     if(a[i]>max)
     {
          max=a[i];
          imax=i; //текущий максимум на месте i
     }
cout<<"Максимальное значение= "<<max<<"\n";
cout<<"Индекс максимума= "<<imax<<"\n";
```

Данный алгоритм будет короче, если обойтись без переменной *max*:

```
int imax=0;
for(int i=1;i<SIZE;i++)
     if(a[i]>a[imax]) imax=i;
cout<<"Максимальное значение= "<<a[imax]<<"\n";
cout<<"Индекс максимума= "<<imax<<"\n";
```

Предположим, что в массиве возможны одинаковые элементы. Поэтому максимальное значение может повторяться. Если нужно найти индекс максимального элемента, то следует внимательно изучить условие задачи: требуется найти индекс первого или последнего максимума?

Алгоритм, приведенный выше, выдаст в результате первое появление максимума в массиве, так как в случае совпадения текущего элемента с максимальным значением *imax* не обновляется (рисунок 5.4).

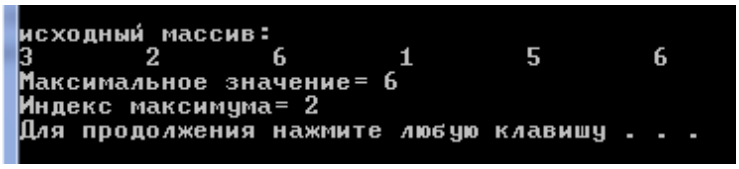


Рисунок 5.4 – **Результат поиска первого максимального элемента**

Чтобы получить последний максимальный элемент, нужно в операторе if поставить условие >=. Тогда в случае совпадения значений текущего элемента и текущего максимума произойдет обновление значения максимального индекса.

Подсчет количества элементов в массиве, удовлетворяющих некоторому условию

Пусть, например, необходимо подсчитать количество ненулевых элементов в массиве.

Надо ввести счетчик количества ненулевых элементов (kol), который будем увеличивать на единицу каждый раз в случае, если встречается ненулевой элемент при просмотре массива:

```
int kol=0;
for(int i=0;i<SIZE;i++)
     if(a[i]!=0) kol++;
cout<<"Количество ненулевых элементов= "<<kol<<"\n";
```

Поиск первого элемента в массиве, удовлетворяющего некоторому условию

Пусть, например, необходимо найти первый нулевой элемент в массиве и вывести его индекс.

В этом случае лучше использовать цикл while, так как не известно, сколько элементов нужно перебрать до момента, когда встретится нулевой элемент. На самом деле существуют две причины закончить просмотр массива:

- если все элементы уже просмотрены, а ноль так и не найден;
- если нашли первый нулевой элемент.

В самом цикле просто увеличивается индекс текущего элемента:

```
int i=0;
while(i<SIZE&&a[i]!=0) i++;
if(i==SIZE)
     cout<<"В массиве нет нулевых элементов\n";
else
     cout<<"Индекс первого нулевого элемента= "<<i<<"\n";
```

Следует помнить о важности порядка условий цикла while: сначала проверяется условие, что индекс находится в пределах массива (i < SIZE), и только если это условие выполняется, происходит обращение к элементу $a[i]$ в условии $a[i]$!=0. Таким образом, никогда не произойдет обращение к памяти за пределами массива.

Эту задачу можно также решить с помощью цикла for и оператора экстренного выхода из цикла break.

5.4. Генератор случайных чисел в языке С

Для заполнения массивов большой размерности произвольными числами удобнее не вводить эти числа с клавиатуры, а генерировать случайным образом. Для этого в библиотеке <stdlib.h> языка С имеется функция rand(). Она генерирует целое случайное число от 0 до 32 767 согласно равномерному закону распределения. Это означает, что вероятность появления любого из чисел от 0 до 32 767 одинаковая.

Рассмотрим использование функции rand() на следующем примере:

```
#include <stdlib.h>
#include <iostream>
using namespace std;
void main()
{
	setlocale(LC_ALL,"rus");
	int x;
	x=rand();
	cout<<"Первое случайное число= "<<x<<"\n";
	x=rand();
	cout<<"Второе случайное число= "<<x<<"\n";
	system("pause");
}
```

Библиотека stdlib.h в Visual Studio подключается автоматически (соответствующий заголовочный файл имеется в папке *Внешние зависимости*), поэтому, в принципе, первую строку этой программы (`#include <stdlib.h>`) можно и опустить. Но для того чтобы программы обладали свойством переносимости, т. е. работали не только в Visual Studio, но и в других средах программирования (или могли компилироваться отдельными компиляторами), лучше все используемые библиотеки подключать командой #include.

Если запустить эту программу один раз, то получим числа 41 и 18 467. Если запустить еще раз, то получим те же числа. Это означает, что генерируемые числа на самом деле не так уж и случайны.

И действительно, на самом деле такие числа называются «псевдослучайными», так как каждое следующее число рассчитывается на основании предыдущего. Поэтому вся цепочка псевдослучайных чисел зависит от начального значения генератора, которое используется для расчета самого первого числа. Изменив начальное значение генератора, изменим и всю цепочку генерируемых в программе псевдослучайных чисел. Сделать это можно с помощью функции srand(int *x*), параметром которой является задаваемое начальное значение генератора.

Например, задав в начале нашей программы функцию srand(5), получим «случайные» числа 54 и 28 693. А если задать srand(7), то получим 61 и 17 422. Но при повторном запуске программы получаем опять те же самые числа, а нам бы хотелось, чтобы при каждом запуске программы числа были разными.

Решение состоит в том, чтобы в качестве начального значения генератора задать текущее время в тот момент, когда программа запускается. Поскольку это время будет каждый раз разное, то и последовательность генерируемых чисел будет другая.

В библиотеке <time.h> (автоматически не подключается) есть функция time(). Она возвращает время (в секундах), прошедшее с 1 января 1970 г.

Функции time() можно передать параметр, являющийся указателем на объект, в который запишется текущее значение времени. Параметр может быть и нулевым указателем: time(NULL) или time(0). В этом случае используется значение функции, например:

```
srand(time(0));
```

В результате начальным значением генератора станет текущее время в секундах. Следовательно, при каждом запуске программы получим разную последовательность псевдослучайных чисел.

Чтобы генерировать целые числа в заданном диапазоне, можно использовать операцию деления по модулю (%). Поскольку остаток от деления всегда меньше делителя, то написав, например, rand()%100, получим случайное число в диапазоне от 0 до 99. Тогда:

```
x=rand()%100+1; //x получит случайное значение от 1 до 100.
```

Выведем теперь формулу в общем виде. Если нужно генерировать числа от a до b, то всего таких чисел должно быть $b - a + 1$. rand()%($b - a + 1$) даст случайные числа от 0 до $b - a$. Добавим смещение на a: rand()% ($b - a + 1$) + a и получим числа от a до b:

```
x=rand()%(b-a+1)+a; // случайное число в диапазоне от a до b.
```

Пример 5.1. Необходимо инициализировать одномерный массив случайными числами от 10 до 20 и вывести элементы массива на консоль.

```
#include <iostream>
using namespace std;
#include <stdlib.h>//б-ка содержит rand() и srand()
#include <time.h>//б-ка содержит функцию time()
#define SIZE 7
void main()
{
     setlocale(LC_ALL,"rus");
     int a[SIZE];
     srand(time(0)); //задать начальное значение генератора
     for (int i = 0; i <SIZE; i++)
     {
          a[i] = rand() % 11 + 10; //числа от 10 до 20
          cout << a[i] <<"\t";
     }
     cout << endl; //перевод курсора
     system("pause");
}
```

5.5. Объявление и инициализация двумерных массивов

Кроме одномерных массивов (с одним индексом) в языке С можно объявить и использовать многомерные массивы (с несколькими индексами). В принципе, количество индексов неограниченно, но человеку очень трудно представить себе (и корректно использовать) массив большей размерности, чем трехмерный. Наиболее же часто используются двумерные массивы.

Можно представить двумерный массив (матрицу) в виде таблицы, имеющей определенное количество строк и столбцов. При этом первый индекс означает номер строки, а второй – номер столбца. При объявлении массива после имени отдельно в квадратных скобках указывается количество строк, а затем количество столбцов:

```
тип_данных имя_массива[количество_строк][количество_столбцов];
```

Например:

```
int a[3][4]; //массив из 3-х строк и 4-х столбцов.
```

В этом массиве строки нумеруются от 0 до 2, а столбцы – от 0 до 3 (рисунок 5.5).

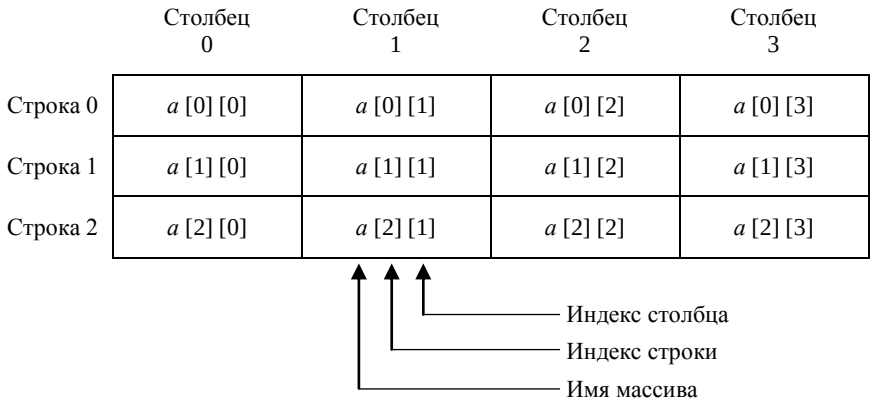


Рисунок 5.5 – **Двумерный массив**

При описании двумерного массива действуют те же правила, что и для массива одномерного: размерность по каждому индексу должна задаваться целочисленным константным значением. Хороший стиль программирования предполагает, что размерность задается с помощью констант:

```
const int ROW=3; //под эти константы
const int COL=4; //выделяется память
int a[ROW][COL];
```

либо так:

```
#define ROW 3        //команда препроцессора
#define COL 4        //место в памяти не резервируется
void main()
{
     int x[ROW][COL]; //Препроцессор до компиляции везде
…                     //вместо ROW и COL подставляет 3 и 4
}
```

Чтобы обратиться к отдельному элементу двумерного массива, нужно после имени указать в квадратных скобках номер строки, а затем (тоже в квадратных скобках) – номер столбца:

```
a[2][1]=10; //присваивание значения элементу во второй строке и первом
            // столбце
```

Двумерные массивы, которые описаны выше, располагаются в стековой или статической памяти. Позднее будет рассмотрен способ описания многомерного массива в динамической памяти (куче). В статической памяти элементы двумерного массива расположены подряд по строкам, т. е. последний индекс всегда изменяется быстрее (рисунок 5.6). Можно считать, что двумерный массив – это массив из элементов, которыми являются строки (массив из массивов).

Рисунок 5.6 – **Размещение двумерного массива в памяти**

Элементы двумерного массива можно инициализировать при объявлении двумя способами:

1. Каждая строка заключается в отдельные фигурные скобки:

```
int a1[2][3]={{1,2,3},{4,5,6}};
```

Массив в результате:

```
1 2 3
4 5 6
```

2. Значения указываются подряд и построчно вписываются в массив:

```
int a1[2][3]={1,2,3,4,5,6};
```

Массив в результате тот же:

```
1 2 3
4 5 6
```

Недостающие элементы в списках инициализируются нулями. Например:

```
int a2[2][3]={1,2,3,4,5};
```

Массив в результате:

```
1 2 3
4 5 0
```

```
int a3[2][3]={{1,2},{4}};
```

Массив в результате:

```
1 2 0
4 0 0
```

Простой способ инициализировать весь массив нулевыми элементами – указать один ноль в списке инициализирующих значений (остальные нули будут добавлены по умолчанию):

```
int a4[2][3]={0};
```

Массив в результате:

```
0 0 0
0 0 0
```

Как и в случае одномерного массива, в двумерном массиве нельзя задать инициализирующих элементов больше, чем определено размерностью (компилятор выдаст ошибку).

Работа с двумерным массивом часто предполагает использование вложенных циклов. Внешний цикл перебирает, например, строки, а внутренний – столбцы в фиксированной строке. Можно и наоборот – в зависимости от постановки задачи.

Примечание – При работе с массивами нужно следить, чтобы не происходило обращение к элементу за пределами памяти, отведенной под массив. Такая ошибка не всегда обнаруживается во время выполнения программы, но может привести к большим проблемам.

5.6. Типовые алгоритмы работы с двумерными массивами

Рассмотрим несколько примеров работы с двумерными массивами.

Пример 5.2. Построчный вывод двумерного массива.

```
const int ROW=3;
const int COL=4;
int a[ROW][COL]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
cout<<"Исходный массив:\n";
for (int i=0;i<ROW;i++) // цикл по строкам
{
     for(int j=0;j<COL;j++) //цикл по столбцам
          cout<<a[i][j]<<"\t";
     cout<<"\n";
}
```

Массив будет выведен так, как показано на рисунке 5.7.

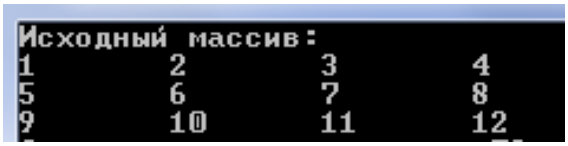


Рисунок 5.7 – **Результат построчного вывода массива**

Пример 5.3. Определение суммы элементов двумерного массива.

```
sum=0;
for (int i=0;i<ROW;i++)     //цикл по строкам
     for(int j=0;j<COL;j++) //цикл по столбцам
          sum+=a[i][j];
cout<<"Сумма элементов массива= "<<sum<<"\n";
```

В этом примере прядок прохождения массива не важен, поэтому возможен и такой вариант:

```
sum=0;
for (int j=0;j<COL;j++)     //цикл по столбцам
     for(int i=0;i<ROW;i++) //цикл по строкам
          sum+=a[i][j];
cout<<"Сумма элементов массива= "<<sum<<"\n";
```

Отметим, что использование переменной *i* в качестве номера строки, а *j* – в качестве номера столбца является традицией. В принципе, можно использовать и другие имена переменных.

Пример 5.4. Определение минимального элемента в каждом столбце двумерного массива.

```
for(int j=0;j<COL;j++)            //цикл по столбцам
{
     min=a[0][j];   //нач.значение минимума - нулевой элемент столбца
     for(int i=1;i<ROW;i++) //цикл по строкам
          if(a[i][j]<min) min=a[i][j];
     cout<<"Минимальный элемент в "<<j<<"-м столбце= "<<min<<"\n";
}
```

Для каждого столбца двумерного массива организуется поиск минимального элемента (как в обычном одномерном массиве), а затем вывод этого значения на консоль. Пример результата выполнения этого алгоритма показан на рисунке 5.8.

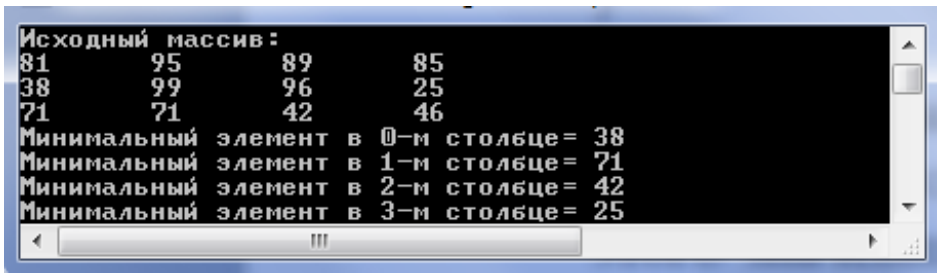


Рисунок 5.8 – **Результат определения минимума в каждом столбце**

Пример 5.5. Уравнения диагоналей.

Часто в задачах с двумерными массивами нужно знать уравнения диагоналей. Например, главная диагональ в квадратной матрице (массив, в котором число строк равно числу столбцов = =SIZE) показана на рисунке 5.9.

Можно заметить, что для любого элемента на этой диагонали номер строки равен номеру столбца: $i = j$.

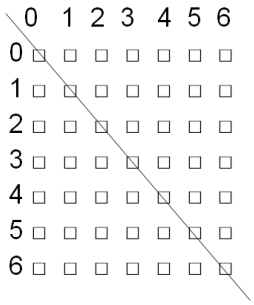


Рисунок 5.9 – **Главная диагональ в квадратной матрице**

Если нужно, например, сложить элементы под главной диагональю (закрашенная часть на рисунке 5.10), то цикл по столбцам будет от 0 до номера строки:

```
sum=0;
for(int i=0;i<SIZE;i++)
     for(int j=0;j<=i;j++)
          sum+=a[i][j];
```

Рисунок 5.10 – **Задача поиска суммы элементов под главной диагональю**

Побочная диагональ квадратной матрицы изображена на рисунке 5.11. Для любого элемента на этой диагонали выполняется соотношение $i + j = \text{SIZE} - 1$.

Отсюда можно получить зависимость номера столбца от номера строки: $j = \text{SIZE} - 1 - i$.

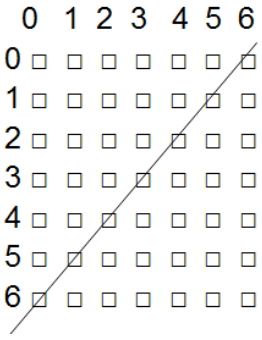


Рисунок 5.11 – **Побочная диагональ матрицы**

Пусть, например, нужно получить сумму элементов матрицы, которые закрашены на рисунке 5.12.

Рисунок 5.12 – **Задача поиска суммы элементов над побочной диагональю**

Индекс столбца во вложенном цикле будет меняться от 0 до побочной диагонали, т. е. до SIZE – 1 – *i*:

```
sum=0;
for(int i=0;i<SIZE;i++)
     for(int j=0;j<=SIZE-1-i;j++)
          sum+=a[i][j];
```

Пример 5.6. Обработка статистики сдачи экзаменов.

В строках двумерного массива представлены результаты сдачи экзаменов (по 100-балльной системе) для 20 студентов группы. Каждый студент сдавал 4 экзамена. Требуется найти:

- самую низкую оценку в группе;
- самую высокую оценку в группе;
- средний балл каждого студента;
- количество студентов, не сдавших сессию (средний балл меньше 50).

Массив оценок следует инициализировать случайными числами.

В начале составим алгоритм решения задачи первого уровня:

1. Инициализация массива оценок.
2. Вывод на консоль информации о каждом студенте (номер, баллы, средний балл) и подсчет количества несдавших экзамен.
3. Определение минимальной и максимальной оценки.
4. Вывод общей статитики по группе: минимальная и максимальная оценки, число студентов, не сдавших сессию.

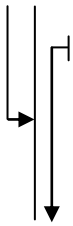
Выполним детализацию этого алгоритма методом нисходящего проектирования. Для подзадачи 1 можем сразу написать программу на языке С:

```
#include <iostream>
#include <time.h>
#define ROW 20
#define COL 4
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     int a[ROW][COL]; //массив оценок
     srand(time(0)); //задание начального значения генератора
     for (int i=0;i<ROW;i++)
          for(int j=0;j<COL;j++)
               a[i][j]=rand()%101; //от 0 до 100 баллов
…
}
```

Вывод информации о всех студентах (подзадача 2) оформим в виде таблицы. Сначала выведем шапку таблицы. Потом в цикле переберем строки двумерного массива (каждая строка соответствует одному студенту). Поскольку в этом цикле еще нужно считать количество несдавших сессию, введем соответствующий счетчик (countFail), который инициализируем нулем перед выполнением цикла по студентам:

Вывести шапку таблицы;
countFail = 0;
***Для** i от 0 до 19*
нц
обработать i-ю строку (студент номер i + 1);
кц

Обработка каждого студента будет заключаться в выводе его оценок на экран и одновременно подсчете суммы оценок. Затем выполняется расчет среднего балла, и он тоже выводится на экран. Если средний балл оказывается меньше 50, то нужно увеличить счетчик несдавших экзамен (count Fail):



Вывести номер студента (i + 1);
Вывести оценки на экран и подсчитать их сумму;
Рассчитать среднюю оценку студента;
Вывести среднюю оценку;
***Если** средняя оценка < 50*
countFail++;
Все-если

Осталось детализировать вывод оценок на экран. Это должен быть цикл по столбцам матрицы оценок (т. е. по всем оценкам студента):

sum = 0;
***Для** j от 0 до 3*
нц
Вывести a[i][j]
sum = sum + a[i][j];
кц

Эта программа на псевдокоде уже дает возможность полностью написать код подзадачи 2:

```
int sum,countFail;   //сумма оценок и количество несдавших
const int POROG=50;  //константа – пороговый средний балл
double average;      //средний балл
cout<<"Исходный массив и средние баллы:\n";
cout<<"№   Mark1   Mark2   Mark3   Mark4   Average\n";
countFail=0;
for (int i=0;i<ROW;i++)     //цикл по студентам
{
     cout<<i+1<<"\t";       //вывод номера студента
     sum=0;
     for(int j=0;j<COL;j++) //цикл по оценкам
     {
          cout<<a[i][j]<<"\t";    //вывод оценок
          sum+=a[i][j];           //подсчет суммы баллов
     }
     average=(double)sum/4; //расчет среднего балла
     if (average<POROG)
          countFail++; //подсчет количества несдавших
     cout<<average<<"\n";         //вывод среднего балла
}
```

Решение подзадачи 3 требует просмотра всего массива оценок. При этом неважно, какой цикл будет внешний: по строкам или по столбцам. Каждую оценку сравниваем с текущим значением максимума и минимума и при необходимости обновляем этот текущий максимум или минимум:

```
int min=a[0][0],max=a[0][0];      //начальные значения максимума и минимума
for(int i=0;i<ROW;i++)            //цикл по студентам
     for(int j=0;j<COL;j++)       //цикл по оценкам
     {
          if (a[i][j]<min) min=a[i][j]; //сравнение с текущим минимумом
          if (a[i][j]>max) max=a[i][j]; //сравнение с текущим максимумом
     }
```

Для решения подзадачи 4 нужно просто вывести те значения, которые получили ранее:

```
cout<<"Минимальный балл= "<<min<<"\n";
cout<<"Максимальный балл= "<<max<<"\n";
cout<<"Количество студентов, не сдавших сессию= "<<countFail<<"\n";
```

5.7. Методы сортировки

Сортировкой называется упорядочивание элементов в массиве данных. *Ключом* сортировки называется поле, по которому происходит упорядочивание.

Обычно в практических задачах в массиве записаны не просто числа, а структуры, объединяющие данные разного типа об одном объекте. Например, массив данных о студентах содержит фамилию студента, его возраст и средний балл. Если отсортировать данные в алфавитном порядке по фамилии, то фамилия – ключ сортировки. Если отсортировать в порядке убывания среднего балла, то ключом будет средний балл и т. д. Поскольку в примерах ниже упорядочиваются числовые массивы, то само значение элемента и будет ключом сортировки.

Различные методы сортировки отличаются скоростью выполнения, затрачиваемой памятью, эффективностью использования для различных наборов данных [1], [2].

Рассмотрим простые методы сортировки:

- метод выбора;
- метод обмена (и его разновидности);
- метод вставок.

Метод выбора

Метод выбора заключается в том, что отсортированная последовательность создается постепенно путем присоединения очередного элемента в нужном порядке. Очередной элемент выбирается из еще не отсортированной части массива.

Различные методы выбора отличаются объектом выбора (минимальный или максимальный элемент) и направлением сортировки (по возрастанию или убыванию).

Пусть, например, необходимо упорядочить массив чисел по возрастанию (неубыванию) методом выбора минимального элемента, т. е. должен быть получен массив, для которого ($a_1 \le a_2 \le \ldots \le a_n$).

Для этого используется следующий алгоритм:

1. В исходном массиве выбирается минимальный элемент.

2. Этот элемент меняется местами с первым элементом массива. Таким образом, один элемент занял свое место в массиве.

3. Далее рассматривается массив без учета этого первого элемента, и в нем повторяются те же действия (выбирается минимальный элемент в части массива, начиная со второго элемента, и меняется местами со вторым элементом).

4. Аналогично процесс продолжается далее, и последний раз минимальный элемент выбирается из двух последних элементов массива.

Пример 5.7. Пусть имеется массив из пяти элементов:

3 6 1 8 2

Выбираем в нем минимальный элемент и меняем местами с первым элементом:

Теперь единица оказалась на своем месте, и можно рассмотреть массив меньшего размера (начиная со второго элемента и до конца):

1 6 3 8 2

В этой части массива также находим минимальный элемент и меняем его местами со вторым элементом массива:

Теперь уже два элемента стоят на своем месте, и можно рассмотреть остаток массива, начиная с третьего элемента:

1 2 3 8 6

Находим минимальный и меняем его местами с третьим элементом (т. е. в данном примере фактически оставляем тройку на своем месте):

1 2 3 8 6

Остается рассмотреть часть массива из двух последних элементов, в котором вновь находим минимальный:

1 2 3 8 6

После обмена его с четвертым элементом получаем упорядоченный массив:

1 2 3 6 8

Таким образом, на каждом этапе рассматривается подмассив, левая граница которого сдвигается вправо до тех пор, пока в этом подмассиве больше чем один элемент.

Введем переменную *k* (левая граница рассматриваемого подмассива). Учитывая, что в языке С элементы массива имеют номера от 0 до *n* – 1 (где *n* – размерность массива), *k* будет изменяться от 0 до *n* – 2. Тогда алгоритм сортировки методом выбора на псевдокоде можно записать так:

Для *k от 0 до n – 2*
нц
    Найти минимум в подмассиве от k до n – 1
    Поменять местами этот минимум и a[k]
кц

Детализируем подзадачу поиска минимума:

```
imin=k;              // считаем первый элемент подмассива минимальным
for(int i=k+1;i<N;i++) // перебираем элементы от k+1 до конца
     if(a[i]<a[imin]) imin=i; //обновляем индекс минимума
```

и подзадачу перестановки двух чисел:

```
tmp=a[k];
a[k]=a[imin];
a[imin]=tmp;
```

Таким образом, полностью алгоритм сортировки методом выбора на языке С будет иметь следующий вид:

```
for(int k=0;k<N-1;k++) // к - левая граница подмассива
{
     imin=k;           // считаем первый элемент подмассива минимальным
     for(int i=k+1;i<N;i++) // перебираем элементы от k+1 до конца
          if(a[i]<a[imin]) imin=i; //обновляем индекс минимума
     //меняем местами a[k] и a[imin]
     tmp=a[k];
     a[k]=a[imin];
     a[imin]=tmp;
}
```

Данный алгоритм не использует никакой дополнительной памяти. Однако при большом количестве элементов массива время его реализации значительно возрастает. Такой алгоритм можно использовать для массивов небольшого размера.

Метод обмена (метод пузырька)

Идея *метода пузырька* – одного из методов обмена – состоит в просмотре массива от начала до конца, в процессе которого сравниваются соседние элементы. Если они стоят не по порядку, то они обмениваются местами. В результате одного просмотра элемент, который должен стоять на последнем месте, там и оказывается («всплывает», как пузырек).

Очевидно, что можно таким образом упорядочивать как по невозрастанию, так и по неубыванию. Можно также просматривать элементы слева направо либо справа налево.

Пусть необходимо упорядочить массив по возрастанию (неубыванию), проходы должны выполняться слева направо (от начала к концу массива).

Алгоритм будет следующий:

1. Сравниваются элементы $a[0]$ и $a[1]$. Если они не упорядочены (т. е. не выполняется условие $a[0] <= a[1]$), то эти элементы меняют местами.

Затем сравниваются элементы $a[1]$ и $a[2]$ и при необходимости переставляются. Так просматриваются все пары соседних чисел. Последняя пара будет $a[n-2]$ и $a[n-1]$.

В результате максимальный элемент окажется на последнем месте.

2. Второй просмотр массива также начинается со сравнения элементов $a[0]$ и $a[1]$. Последней будет пара $a[n-3]$ и $a[n-2]$ (поскольку последний элемент массива уже стоит на своем месте). В результате на месте предпоследнего элемента окажется второй по величине элемент («всплыл» второй пузырек).

3. Продолжить просмотры, уменьшая количество рассматриваемых элементов. При последнем просмотре будут сравниваться только элементы $a[0]$ и $a[1]$.

Пример 5.8. Пусть необходимо отсортировать массив из пяти элементов:

6 3 7 1 5

Выполним первый проход по массиву, сравнивая пары соседних элементов (первый элемент из пары отмечен кружком). В случае неупорядоченности пары производим обмен:

⑥3 7 1 5

3 ⑥7 1 5

3 6 ⑦1 5

3 6 1 ⑦5

3 6 1 5 7

В результате первого прохода наибольший из элементов (семерка) оказался на своем месте. При втором проходе этот элемент исключается из рассмотрения (уже стоит на своем месте). Поэтому сравнений будет меньше:

③6 1 5 7

3 ⑥1 5 7

3 1 ⑥5 7

3 1 5 6 7

Аналогично делаем третий проход:

И хотя массив уже упорядочен, согласно данному алгоритму необходимо сделать еще один проход:

Обозначим через *k* правую границу рассматриваемого подмассива. В первом проходе она составляла *n* – 1 (т. е. рассматривался весь массив), во втором проходе будет *n* – 2 (массив без одного элемента) и т. д. В последнем проходе рассматривалась только одна пара, т. е. *k* уменьшается от *n* – 1 до 1.

Индекс первого элемента из пары меняется соответственно от 0 до *k* – 1. Поэтому можно записать этот алгоритм на псевдокоде так:

Для *k от n – 1 до 1*
нц
    Просмотреть пары с индексом первого элемента от 0 до k – 1
кц

Детализируем подзадачу «просмотреть пары»:

Для *i от 0 до k – 1 //i – индекс первого элемента из пары*
нц
    Если *a[i] > a[i + 1] //два соседних элемента не по порядку*
    то
        переставить местами a[i] и a[i + 1]
    Все-если
кц

Подставим фрагменты алгоритма на свое место и перепишем на языке С:

```
for (int k=N-1;k>0;k--) //меняется правая граница подмассива
     for(int i=0; i<k; i++) //меняется номер первого элемента из пары
          if(a[i]>a[i+1])
          {    //перестановка соседних элементов
               tmp=a[i];
               a[i]=a[i+1];
               a[i+1]=tmp;
          }
```

Минимизация числа просмотров при сортировке методом пузырька. При сортировке методом пузырька часто встречается ситуация, когда массив уже отсортирован, а просмотры продолжаются. Чтобы вовремя прекратить процесс сортировки, заведем переменную-флаг, которая будет фиксировать факт перестановки каких-либо элементов:

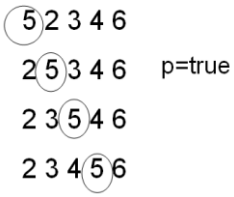
```
bool p; // true, если была перестановка при очередном проходе
```

Переменной *p* присваивается false перед началом прохода по массиву. Если по окончании прохода эта переменная осталась false, то перестановок не было (массив уже упорядочен). В этом случае новой итерации цикла проходов уже не нужно.

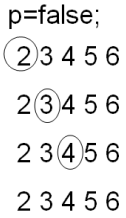
Например, пусть дан массив:

5 2 3 4 6

Перед началом прохода по массиву *p* = false; выполняем проход:



При сравнении 5 и 2 была выполнена перестановка, и переменная *p* получила значение true (на самом деле она получала значение true несколько раз – при каждой перестановке, но нам достаточно и одного раза). Поскольку после первого прохода *p* == true (т. е. перестановки были), то начинаем новый проход:



В процессе прохода не было ни одной перестановки, и флаг *p* остался false. Следовательно, массив уже упорядочен, и дальнейшие просмотры следует прекратить.

Чтобы реализовать такой алгоритм, лучше использовать цикл while для контроля за проходами:

```
bool p; // true, если обмены были
int k=N-1;
//пока не закончены плановые проходы и обмены были
while(k>0&&p)
{
     p=false; //сброс флага перед проходом
     for(int i=0; i<k; i++) //меняется номер первого элемента из пары
          if(a[i]>a[i+1])
          {    //перестановка соседних элементов
               tmp=a[i];
               a[i]=a[i+1];
               a[i+1]=tmp;
               p=true; //установка флага: обмены были
          }
}
```

Другой вариант реализации этого алгоритма – цикл for и экстренный выход из цикла с помощью break.

Модифицированный метод пузырька. Еще один способ усовершенствовать метод пузырька – запомнить индекс последнего обмена при очередном просмотре. Тогда следующий просмотр может быть только до этой позиции (правее все элементы уже стоят по порядку).

Обозначим *posl* индекс первого элемента из последнего обмена.

Если перед началом очередного просмотра задать *posl* = –1, то эту переменную можно использовать также и в качестве флага о том, что обмены были. Если обменов не было при очередном просмотре, то posl останется –1.

Если же обмены были, то posl покажет индекс последнего обмена. Тогда следующий просмотр можно делать уже до этого индекса:

```
int posl; //индекс последнего обмена
int k=N-1;
//пока не закончены плановые проходы и обмены были
while(k>0)
{
     posl=-1; //флаг: обменов не было
     for(int i=0; i<k; i++) //меняется номер первого элемента из пары
          if(a[i]>a[i+1])
          {    //перестановка соседних элементов
               tmp=a[i];
               a[i]=a[i+1];
               a[i+1]=tmp;
               posl=i; //запоминаем индекс последнего обмена
          }
     k=posl; //следующий просмотр до последнего обмена
}
```

Шейкер-сортировка. Для ускорения процесса сортировки можно к идее предыдущего алгоритма добавить изменение направления проходов на каждом шаге. То есть в первом проходе, например, просмотры слева направо («всплывают» максимальные элементы), а во втором проходе – справа налево («всплывают» минимальные элементы). И так каждый раз меняется направление.

Для реализации этой идеи можно использовать две переменные:

- *low* – нижняя граница сортируемого подмассива (начинается с 0);
- *up* – верхняя граница сортируемого подмассива (начинается с $n - 1$).

Тогда смысл продолжать сортировку возникнет в том случае, если нижняя и верхняя граница «не перехлестнулись», т. е. выполняется условие $low < up$:

```
while(low<up)
{
     //проход слева направо
     //проход справа налево
}
```

Полностью код программы шейкер-сортировки следующий:

```
#include <iostream>
using namespace std;
#include <time.h>
void main() {
     setlocale(LC_ALL, "rus");
     srand(time(0));
     const int N = 10;
     int a[N];
     //инициализация и вывод массива
     cout<<"Исходный массив:\n";
     for (int i = 0; i < N; i++) {
          a[i] = rand() % 21;
          cout<< a[i] <<" ";
     }
     cout<< endl;
     int low = 0, up = N - 1;
     int posl,tmp;
     while (low < up) {
          //проход слева направо
          posl = -1; //сброс индекса последнего обмена
```

```
            for (int i = low; i < up; i++) {
                 if (a[i] > a[i + 1]) {
                      tmp = a[i];
                      a[i] = a[i + 1];
                      a[i + 1] = tmp;
                      posl = i; //запоминаем индекс последнего обмена
                 }
            }
            up = posl; //подвигаем правую границу
            //сортируемой области к последнему обмену
            //проход справа налево
            posl = N;//если обменов не будет,
            //то останется такое невозможное значение
            for (int i = up - 1; i >= low; i--) {
                 if (a[i] > a[i + 1]) {
                      tmp = a[i];
                      a[i] = a[i + 1];
                      a[i + 1] = tmp;
                      posl = i;// запоминаем индекс последнего обмена
                 }
            }
            low = posl + 1; //подвигаем левую границу
            //к последнему обмену
       }
       //вывод отсотрированного массива
       cout<<"Отсортированный массив:\n";
       for (int i = 0; i < N; i++) {
            cout << a[i] <<" ";
       }
       cout << endl;
       system("pause");
  }
```

Метод вставок

Идея метода вставок состоит в том, что у нас всегда есть два подмассива: один отсортированный (в самом начале он состоит из одного первого элемента), а второй – еще не отсортированный. Из начала второго массива берем первый элемент и вставляем его в отсортированную часть массива так, чтобы порядок сохранился. И так продолжается до тех пор, пока все элементы не будут перемещены из неотсортированной части в отсортированную.

Пусть, например, по неубыванию упорядочены элементы от $a[0]$ до $a[i]$ $a[0] \le a[1] \le \ldots \le a[i]$, и нужно вставить элемент $a[i + 1]$ так, чтобы последовательность элементов $a[0] \ldots a[i + 1]$ осталась упорядоченной.

Этот процесс нужно повторять, пока не вставим все элементы массива, т. е. i меняется от 0 до $n - 2$.

Процесс вставки будем проводить, «погружая» вставляемый элемент справа налево до тех пор, пока он не займет свое место.

Для этого введем вспомогательную переменную *tmp*, в которой сохраним значение вставляемого элемента. Также введем текущий индекс движения влево: сначала j равен i, а затем уменьшается (использовать переменную i нельзя – она нужна, чтобы продолжать выбор неотсортированных элементов). Проверяем условие $a[j] > tmp$ и в случае его выполнения подвигаем j-й элемент вправо (освобождаем место для вставки). После чего уменьшаем индекс j.

Процесс продолжается до левого края массива или до нахождения места, подходящего для вставляемого значения.

Пример 5.10. Пусть часть массива уже упорядочена (выделена серым):

1 2 7 8 ⑤ 9 6

Нужно взять очередной элемент из упорядоченной части (пятерку) и вставить его на нужное место.

Индекс последнего элемента из упорядоченных равен 3. Поэтому начальное значение вспомогательного индекса $j = 3$. Во временную переменную записываем значение вставляемого элемента: $tmp = 5$.

Далее сравниваем значение вставляемого с текущим (j-м). Поскольку вставляемый элемент меньше ($a[j] > tmp$), сдвигаем восьмерку вправо:

1 2 7 8 8 9 6

и уменьшаем текущий индекс ($j = 2$). Далее сравниваем вставляемую пятерку с числом 7. И опять место еще не найдено ($a[2] > tmp$). Сдвигаем:

1 2 7 7 8 9 6

Уменьшаем индекс: $j = 1$. Поскольку условие $a[j] > tmp$ нарушается, место для вставляемого элемента найдено. Записываем его на место $j + 1$:

1 2 5 7 8 9 6

Таким образом, получили упорядоченный массив большего размера, чем предыдущий.
Запишем теперь этот алгоритм на псевдокоде:

```
Для i от 0 до n – 2 //i – индекс последнего из упорядоченных элементов
нц
      Элемент a [i + 1] вставить на нужное место от 0 до i + 1
кц
```

Детализируем процесс вставки:

```
tmp = a [i + 1]; //сохранить копию вставляемого элемента;
j = i; // j – индекс элемента, после которого может быть вставка;
Пока j >= 0 и a [j] > tmp //не вышли за пределы и не найдено место;
нц
      сдвинуть a [j] на место j + 1;
      j– –;
кц
Вставить tmp на место j + 1.
```

После соединения фрагментов алгоритма получим следующую программу на языке С:

```
for(int i=0;i<N-1;i++) //i-номер последнего из упорядоченных
{
     tmp=a[i+1]; //сохранить копию вставляемого элемента
     j=i; //j- индекс элемента, после которого может быть вставка
     while(j>=0&&a[j]>tmp) //пока не вышли за пределы и не найдено место
     {
          a[j+1]=a[j]; //сдвиг элемента вправо
          j--;
     }
     a[j+1]=tmp; //вставка на место
}
```

5.8. Методы поиска

Существует достаточно много алгоритмов поиска индекса нужного элемента в массиве. Рассмотрим только два из них: линейный и бинарный.

Линейный поиск может применяться в любом массиве (в том числе и в неотсортированном). Суть его состоит в переборе элементов массива до тех пор, пока не будет найден нужный элемент, например:

```
int i;
for (i=0; i<N; i++)
     if (a[i]==elem) break;
if (i==N)
     cout<<"Искомый элемент не найден\n";
else
     cout<<"Искомый элемент на месте "<<i+1<<"\n";
```

Бинарный (двоичный) поиск применяется в отсортированном массиве. Использование того факта, что массив отсортирован, позволяет значительно ускорить процесс поиска.

Допустим, что массив упорядочен по неубыванию.

Суть бинарного поиска состоит в том, что массив разбивается пополам и искомое значение сравнивается со серединным элементом. Если искомое значение меньше элемента в середине массива, то дальнейший поиск нужно проводить в левой части массива (поскольку справа все элементы заведомо будут больше). Если же искомое значение больше серединного элемента, то продолжается поиск в правой части. Таким образом, за один шаг алгоритма область поиска сужается в два раза.

Процесс продолжается делением оставшейся части массива пополам и выполнением тех же действий до тех пор, пока не будет найден искомый элемент либо границы «перехлестнутся».

Обозначим границы области поиска: сначала *low* = 0 – нижняя граница и *up* = *n* – 1 – верхняя граница. Поиск продолжается пока *low* <= *up*; *middle* – серединный элемент в рассматриваемой части массива (с учетом округления).

Переменная *find* хранит индекс найденного элемента. До начала поиска ей присваивается значение –1. Если по окончании процесса оно таким и останется, то поиск не дал результатов (в массиве нет такого значения).

```
int low=0, up=N-1, middle;
int find=-1; //индекс найденного элемента
do
{
     middle=(low+up)/2; //серединный элемент
     if(elem==a[middle]) //элемент найден
     {
          find= middle;
          break;
     }
     if(elem<a[middle]) up=middle-1; //ищем далее в левой половине
     if(elem>a[middle]) low=middle+1; //ищем далее в правой половине
}
while(low<=up);
if (find==-1)
     cout<<"Искомый элемент не найден\n";
else
     cout<<"Искомый элемент на месте "<<find+1<<"\n";
```

Задачи

Одномерные массивы

Задача 5.1. Ввести массив целых чисел из шести элементов. Найти количество нулей в массиве и сумму элементов, стоящих на нечетных местах.

Задача 5.2. Массив из десяти элементов заполнить случайными целыми числами от 0 до 20. Найти среднее арифметическое элементов массива и количество элементов, меньших этого среднего.

Задача 5.3. Массив из десяти элементов заполнить случайными целыми числами от –10 до 10. Поменять местами первый отрицательный и последний положительный элементы.

Задача 5.4. Ввести массив из восьми элементов. Сформировать новый массив из отрицательных элементов первого массива.

Задача 5.5. Ввести массив, упорядоченный по неубыванию (проверить факт упорядоченности). Найти сумму положительных элементов массива. Использовать факт упорядоченности (положительные элементы в таком массиве расположены в конце, поэтому не нужно перебирать все элементы, а просуммировать только положительные, начиная с конца).

Для самостоятельного решения

Задача 5.6. Инициализировать массив случайными числами. Вывести значение элементов массива на экран.

Задача 5.7. Найти последний максимальный элемент в массиве. Распечатать его значение.

Задача 5.8. Найти первый минимальный элемент в массиве. Распечатать его номер.

Задача 5.9. Найти сумму элементов массива.

Задача 5.10. Найти произведение элементов массива, стоящих на нечетных местах.

Задача 5.11. Найти среднее арифметическое элементов массива.

Задача 5.12. Найти номер первого отрицательного элемента либо распечатать сообщение о его отсутствии.

Задача 5.13. Распечатать значение последнего отрицательного элемента либо сообщение о его отсутствии.

Задача 5.14. Подсчитать количество нулевых элементов в массиве.

Задача 5.15. Ввести элементы массива и вывести их в обратном порядке.

Задача 5.16. Ввести массив целых чисел и границы интервала $[a, b]$. Найти среднее арифметическое чисел, не попадающих в промежуток $[a, b]$, и количество положительных чисел, стоящих на местах, кратных 3.

Задача 5.17. Найти в массиве последний максимальный элемент и поменять его местами с первым минимальным элементом массива.

Задача 5.18. Ввести массив, упорядоченный по невозрастанию, и число A. Определить, содержится ли в массиве значение A и, если да, определить индекс первого такого элемента.

Задача 5.19. Найти в массиве значения, которые повторяются два и более раз, и вывести их на экран.

Задача 5.20. Найти в массиве самое маленькое нечетное число.

Задача 5.21. Сформировать одномерный массив из десяти элементов, заполнив его случайными числами от 5 до 15. Определить местоположение первого минимального элемента и последнего максимального элемента. Поменять эти элементы местами.

Задача 5.22. Сформировать одномерный массив из шести элементов, организовав ввод с консоли. Определить среднее арифметическое значение отрицательных элементов (если в массиве имеется хотя бы один) либо вывести сообщение, что таких элементов нет.

Задача 5.23. Создать массив из восьми элементов и инициализировать его положительными и отрицательными значениями. Найти первый отрицательный элемент и поменять его местами с последним положительным элементом.

Задача 5.24. Сформировать одномерный массив из пятнадцати элементов, заполнив его случайными числами от –20 до 20. Найти сумму отрицательных элементов и произведение элементов массива, стоящих на четных местах.

Задача 5.25. Сформировать одномерный массив из восьми элементов, организовав ввод с консоли. Выполнить реверс массива (первый элемент поменять местами с последним, второй – с предпоследним и т. д.).

Задача 5.26. Сформировать одномерный массив из десяти элементов, заполнив его случайными числами от 0 до 5. Уплотнить массив, удалив из него все нулевые элементы. Распечатать количество элементов, которые были удалены.

Задача 5.27. Сформировать массив из десяти элементов, заполнив их случайными числами от 0 до 50. Определить среднее арифметическое значение элементов массива. Вывести элементы, значение которых оказывается меньше среднего.

Задача 5.28. Ввести массив из восьми элементов. Найти сумму тех элементов массива, которые стоят после элемента, введенного пользователем.

Двумерные массивы

Задача 5.29. Элементы двумерного массива размерностью 3 на 4 инициализировать целыми случайными числами от 10 до 30. Вывести исходный массив. Затем обнулить вторую строку и вывести измененный массив.

Задача 5.30. Элементы массива размерностью 2 на 6 инициализировать целыми случайными числами от 0 до 5. Для каждой строки распечатать индекс первого нулевого элемента (или слово «нет»).

Задача 5.31. Элементы квадратной матрицы размерностью 5 на 5 инициализировать целыми случайными числами от 0 до 10. Найти сумму максимальных элементов всех столбцов.

Задача 5.32. Элементы массива размерностью 4 на 3 инициализировать случайными целыми числами от –5 до 5. Подсчитать количество строк, не содержащих ни одного нулевого элемента.

Задача 5.33. Дана квадратная матрица порядка n (n строк и n стобцов). Найти наибольшее из значений элементов, расположенных в закрашенной части матрицы (рисунок 5.13).

Рисунок 5.13 – **Задача поиска максимума в части двумерного массива**

Для самостоятельного решения

Задача 5.34. Ввести двумерный массив размерностью 2 на 3. Найти количество нулевых элементов в этом массиве.

Задача 5.35. Элементы квадратной матрицы размерностью 6 на 6 инициализировать случайными числами от –20 до 20. Подсчитать и распечатать количество нулевых элементов в каждом столбце.

Задача 5.36. Элементы двумерного массива размерностью 3 на 4 ввести с клавиатуры. Найти номер первой из строк, не содержащей ни одного положительного элемента.

Задача 5.37. Дана квадратная матрица порядка *n* (*n* строк и *n* стобцов). Найти сумму значений элементов, расположенных в закрашенной части матрицы (рисунок 5.14).

Рисунок 5.14 – **Задача поиска суммы элементов в части двумерного массива**

Сортировка и поиск

Задача 5.38. Написать программу сортировки одномерного массива целых чисел по невозрастанию (убыванию) методом выбора максимального элемента.

Задача 5.39. Написать программу, выполняющую сортировку одномерного массива целых чисел по возрастанию методом пузырька; направление просмотров – слева направо. Минимизировать число просмотров.

Задача 5.40. Написать программу, выполняющую сортировку одномерного массива целых чисел по убыванию методом вставок.

Задача 5.41. Дан массив случайных чисел в диапазоне от –20 до 20. Найти позиции самого левого отрицательного элемента и самого правого отрицательного элемента и отсортировать элементы между ними по убыванию.

Задача 5.42. Дан массив случайных чисел от 0 до 20. Пользователь вводит число. Необходимо найти самую левую позицию этого числа в массиве. Отсортировать элементы справа от найденного по возрастанию, а слева – по убыванию.

Задача 5.43. Ввести два одномерных массива из пяти элементов. Объединить их значения и отсортировать полученный массив методом вставок.

Задача 5.44. Написать программу, выполняющую сортировку одномерного массива целых чисел по невозрастанию методом выбора минимального элемента.

Задача 5.45. Написать программу, выполняющую сортировку одномерного массива целых чисел по убыванию методом пузырька. Направление просмотров – справа налево.

Задача 5.46. В упорядоченном массиве символов осуществить поиск символа, введенного пользователем, методами линейного и бинарного поиска (вывести его индекс).

Задача 5.47. Написать программу «справочник». Создать два одномерных массива. Один массив хранит номера ICQ, второй – телефонные номера. Реализовать меню для пользователя:

- отсортировать по номерам ICQ;
- отсортировать по номерам телефона;
- вывести список пользователей;
- выход.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. **Кнут, Д.** Искусство программирования : в 3 т. / Д. Кнут. – 2-е изд. – М. : Вильямс, 2007. – Т. 3 : Сортировка и поиск. – 824 с.

2. **Алгоритмы**: построение и анализ / Т. Х. Кормен [и др.]. – 2-е изд. – М. : Вильямс, 2006. – 1296 с.

3. **Прата, С.** Язык программирования С. Лекции и упражнения / С. Прата. – 6-е изд. – М. : Вильямс, 2015. – 928 с.

4. **Дейтел, Х.** Как программировать на С / Х. Дейтел, П. Дейтел. – 7-е изд. – М. : Бином, 2015. – 1008 с.

5. **Керниган, Б.** Язык программирования С / Б. Керниган, Д. Ритчи. – 2-е изд. – М. : Вильямс, 2016. – 288 с.

6. **Павловская, Т. А.** С/С++. Процедурное и объектно-ориентированное программирование : учеб. / Т. А. Павловская. – СПб. : Питер, 2015. – 496 с.

7. **Шилдт, Г.** С++. Базовый курс / Г. Шилдт. – 3-е изд. – М. : Вильямс, 2015. – 624 с.

8. **Культин, Н. Б.** С/С++ в задачах и примерах / Н. Б. Культин. – 2-е изд. – СПб. : БХВ-Петербург, 2012. – 368 с.

СОДЕРЖАНИЕ

Учебное издание

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Пособие
для реализации содержания образовательных программ
высшего образования I ступени

В двух частях

Часть 1

Автор-составитель
Еськова Оксана Ивановна

Редактор Е. В. Седро
Компьютерная верстка Л. Ф. Барановская

Подписано в печать 06.03.18. Формат 60 × 84 $^{1}/_{8}$.
Бумага офсетная. Гарнитура Таймс. Ризография.
Усл. печ. л. 13,48. Уч.-изд. л. 11,31. Тираж 50 экз.
Заказ №

Издатель и полиграфическое исполнение:
учреждение образования «Белорусский торгово-экономический
университет потребительской кооперации».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий
№ 1/138 от 08.01.2014.
Просп. Октября, 50, 246029, Гомель.
http://www.i-bteu.by

БЕЛКООПСОЮЗ
УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БЕЛОРУССКИЙ ТОРГОВО-ЭКОНОМИЧЕСКИЙ
УНИВЕРСИТЕТ ПОТРЕБИТЕЛЬСКОЙ КООПЕРАЦИИ»

Кафедра информационно-вычислительных систем

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Пособие
для реализации содержания образовательных программ
высшего образования I ступени

В двух частях

Часть 1

Гомель 2018